Speculative Multithreading Does not (Necessarily) Waste Energy

Draft paper submitted for publication. November 6, 2003.

Please keep confidential

Jose Renau, Smruti Sarangi, James Tuck, Karin Strauss, Luis Ceze, Wei Liu, Josep Torrellas
University of Illinois

Abstract

While Chip Multiprocessors (CMP) with Speculative Multithreading (SM) have been gaining momentum, experienced processor designers in industry have reservations about their practical implementation. In particular, it is felt that SM is too energy-inefficient to compete against conventional superscalars.

This paper challenges the commonly-held view that SM consumes excessive energy. We show a CMP with SM support that is not only faster but also more energy efficient than a state-of-the-art wide-issue superscalar. We demonstrate it with a new energy-efficient CMP micro-architecture. In addition, we identify the additional sources of energy consumption in SM, and propose energy-centric optimizations that mitigate them. Experiments with the SpecInt 2000 codes show that a CMP with 2 4-issue cores and support for SM delivers a speedup of 1.08 over a 8-issue superscalar and consumes only 54% of its power. Alternatively, for the same average power in both chips, the SM CMP is 1.6 times faster than the superscalar on average.

1 Introduction

Substantial research effort is currently being devoted to speeding up hard-to-parallelize non-numerical applications such as SpecInt. Designers build sophisticated out-of-order processors, with carefully-tuned execution engines and memory subsystems. Unfortunately, these systems tend to combine high design complexity with diminishing performance returns, which has motivated the search for design alternatives.

One such alternative is Speculative Multithreading (SM) on a Chip Multiprocessor (CMP) [2, 7, 8, 9, 10, 14, 15, 18, 19, 24]. Under SM, these hard-to-analyze applications are carefully partitioned into tasks, which are then optimistically executed in parallel, hoping that no data or control dependence will be violated. A hardware safety net monitors the tasks' control flow and data accesses, watching for violations at run time. When one happens, the hardware transparently rolls back the incorrect tasks and, after repairing the state, restarts them.

SM on a CMP has been the subject of intense study for nearly a decade now. Recent results appear to show that a few processors on a CMP with SM support can speed up hard-to-parallelize non-numerical applications as much as or more than wider-issue superscalars (e.g. [2, 8, 9, 22]).

This is significant because CMPs are attractive platforms: unlike wide-issue superscalars, they provide a decentralized architecture with low-complexity processors. Moreover, CMPs have a natural advantage for explicitly-parallel codes.

Unfortunately, experienced processor designers in industry have reservations about the practical implementation of SM. In particular, it is felt that SM is too energy-inefficient to seriously challenge superscalars. The rationale is that aggressive speculative execution of possibly unnecessary or incorrect tasks is not the best course in a day and age when processors are primarily constrained by energy issues.

Indeed, energy issues have become arguably the main concern for designers of high-end microprocessors. Energy and power consumption directly affect the cost of powering and cooling the system, influence the reliability and aging characteristics of chips, and determine battery life in portable devices. While the simpler cores in a CMP are energy-efficient, CMPs with SM will not be accepted unless their overall energy requirements are competitive against wide-issue superscalars.

In this paper, we directly address the problem of energy consumption in SM. Our main contribution is to show that contrary to popular belief, SM does not necessarily waste energy. We show that SM is not only faster, but also more energy efficient than a state-of-the-art wide-issue superscalar. We demonstrate it with a new energy-efficient micro-architecture for a CMP with SM. This is the first paper to show that SM on a CMP is an interesting design point even for high-performance power-constrained designs.

In addition, we identify and analyze the sources of energy consumption in SM. These issues are: the wasted work of squashed tasks, storage and logic in the memory hierarchy to support data versioning, additional traffic in the memory subsystem, and additional instructions.

We also propose energy-centric optimizations that mitigate these SM sources of energy consumption. These optimizations have been overlooked in performance-centric SM designs because they enhance energy-savings and not performance.

In our experiments with the SpecInt 2000 benchmarks, we show that a CMP with 2 4-issue cores delivers a speedup of 1.08 over an 8-issue superscalar while consuming only 54% of its power. Alternatively, for the same average power in both chips, the SM CMP is 1.6 times faster than the superscalar on average.

This paper is organized as follows: Section 2 provides background on SM; Section 3 analyzes the sources of energy consumption in SM and proposes energy-centric optimizations; Section 4 describes the proposed SM CMP micro-architecture; Section 5 describes our SM compilation infrastructure; Sections 6 and 7 present our evaluation methodology and the evaluation; and Section 8 concludes.

2 Speculative Multithreading

Overview. Speculative multithreading (SM) consists in extracting tasks from a sequential code and executing them in parallel, hoping not to violate sequential semantics. The control flow of the

sequential code imposes a task order and, therefore, we use the terms predecessor and successor tasks. The safe (or non-speculative) task precedes all speculative tasks. The sequential code also yields a data dependence relation on the memory accesses issued by the different tasks, which parallel execution cannot violate. As tasks execute, special hardware support checks that no cross-task dependence is violated. If any is, the incorrect tasks are squashed, any polluted state is repaired, and the tasks are re-executed.

Cross-Task Dependence Violations. Data dependences are typically monitored by tracking, for each individual task, the data written and the data read with exposed reads. An exposed read is a read that is not preceded by a write to the same location. A data dependence violation occurs when a task writes a location that has been read by a successor task with an exposed read. A control dependence violation occurs when a task is spawned in a mispredicted branch path. Dependence violations lead to task squashes, which involve discarding the work produced by the task. Squashes come in two forms. In a control violation, the task is squashed with kill signal. In a data violation, the task is squashed with a restart signal, which also restarts the task from its beginning, hoping that the re-execution will not violate the data dependence.

State Buffering. Memory accesses issued by a speculative task must be handled carefully. Stores generate speculative state that cannot be merged with the safe state of the program. The reason is that it may be incorrect. Consequently, the state is stored separately, typically in the cache of the processor running the task. If a violation is detected, the state generated by the task is discarded. Otherwise, when the task becomes non-speculative, the state is allowed to propagate to memory. When a non-speculative task finishes execution, it commits. Committing informs the rest of the system that the state generated by the task is now part of the safe program state. Commit is done in task order and involves passing a commit token between tasks.

Data Versioning. A task has at most a single version of any given variable. However, different speculative tasks that run concurrently in the machine may write to the same variable and, as a result, produce different versions of the variable. Such versions must be buffered separately. Moreover, when a speculative task reads, it needs to be provided with the closest predecessor version of the variable. Finally, as tasks commit in order, data versions need to be merged with the safe memory state also in order.

Multi-Versioned Caches. A cache that can hold state from multiple tasks is called multi-versioned. There are two performance reasons why multi-versioned caches are desirable: they avoid processor stall when tasks are imbalanced, and enable lazy commit.

If tasks have load imbalance, a processor may finish a task and the task still be speculative. If the cache can only hold state for a single task, the processor has to stall until the task commits. An alternative is to move the task's state to some other buffer, but this complicates the design. Instead, we want the cache to retain the state from the old task and allow the processor to execute another task. If so, the cache has to be multi-versioned.

Lazy commit [11] is an approach where, when a task commits, it does not eagerly merge its

cache state with main memory through ownership requests [15] or write backs [9]. Instead, the task simply passes the commit token to its successor. Its state remains in the cache and is lazily merged with main memory later, usually as a result of cache line replacements. This approach improves performance because it speeds up the commit operation. However, it requires multi-versioned caches.

Tagging Multi-Versioned Caches. Multi-versioned caches typically require that we tag each cache line with a version ID, which records what task the line belongs to. Intuitively, such version ID could be the global ID of the task. Unfortunately, the ID of a task can be quite long. Consequently, to save space, it is best to translate global task IDs into some arbitrary Local IDs (LIDs) that are much shorter. These LIDs are used only locally in the cache, to tag cache lines. This type of ID indirection was first used by Steffan *et al.* [17].

While these LIDs save space in the tags, they need to be translated. They can be kept in a small, per-cache table that we call LID Table. Each cache has a different LID Table.

2.1 Architecture and Environment Considered

SM can be supported in different ways. In this paper, we focus on a Chip Multiprocessor (CMP) architecture because it is a decentralized, potentially energy-efficient platform. To reduce non-commodity hardware, we assume that the processors in the CMP can only communicate via the memory system — there is no hardware support for register communication. In addition, to gain flexibility, the speculative tasks are generated in software by a SM compiler. This is a new compiler that we recently built. Finally, we concentrate on SpecInt 2000 applications, as these non-numerical applications are hard to speed up with conventional platforms.

3 Speculative Multithreading on a CMP: Energy Cost & Optimization

Supporting SM adds several sources of energy consumption to a CMP. In this section, we analyze these sources and then propose "energy-centric" optimizations that directly mitigate them.

3.1 The Energy Cost of Speculative Multithreading

Given a CMP that supports SM, we identify four main SM-specific sources of energy consumption (Table 1). They are: the wasted work of tasks that are squashed; storage and logic in the memory hierarchy to support data versioning; additional traffic in the memory subsystem; and additional instructions.

3.1.1 Task Squashing

A source of energy consumption specific to SM is the work performed by tasks that ultimately get squashed. Note, however, that not all such work is necessarily wasted. Specifically, in a data dependence violation, a task is squashed and often restarted on the same processor. The new instance of the task can leverage branch prediction training from the previous instance. More importantly, the hardware should allow the new instance to reuse the non-dirty state left in the

Sources of Energy Consumption in SM	Proposed Energy-Centric Optimizations
Task Squashing	Stall a task after its second restart Energy-aware task pruning by profiling
Storage and logic in the memory hierarchy to support data versioning	Avoid eagerly "walking" the cache tags Low-energy reuse of cached data on task restart
Additional traffic in the memory subsystem	——
Additional instructions	——

Table 1: Sources of energy consumption in a CMP that are specific to SM, and energy-centric techniques to mitigate them.

cache by the previous instance. As a result, the new execution can be faster than the previous one.

Task squashing also consumes energy in two other operations: sending the squash signal to the processor where the incorrect task is running, and possibly executing some re-initialization code on that processor. Such code may involve restoring the register state, but does not require accessing any large chunk of data in the caches. Given the low frequency of squashes, the energy consumed in these two operations is negligible.

3.1.2 Storage and Logic for Data Versioning in the Memory Hierarchy

Another source of energy consumption in SM is the additional storage and logic in the memory hierarchy required to support data versioning. In many proposed SM schemes, caches are multi-versioned (e.g. [3, 15]), which means that individual caches can hold state from multiple tasks (Section 2). In this case, each cache line is typically tagged with a version ID, which identifies the task it belongs to. Moreover, messages between caches also include the requesting task's ID. On an external access to a cache, the version ID of an address-matching line in the cache is compared to the ID of the requesting task in the incoming message. From the comparison, the cache may determine that a violation occurred. Overall, supporting data versioning can require extra storage in caches to tag lines with a version ID, and extra logic to compare versions when communication occurs. The details of the memory hierarchy that we use are shown in Section 4.

3.1.3 Additional Traffic in the Memory Subsystem

A SM system generates a higher number of messages than a conventional system. While some of these extra messages are simply the result of parallel execution, there are three SM-specific reasons for the increased message volume.

One reason is that caches do not work as well. Caches often have to retain lines from older tasks that ran on the processor and are still speculative. Only when such tasks become safe can the data be displaced. As a result, there is less space in the cache for data that may be useful to the task currently running locally. This higher cache pressure increases displacements of useful lines and subsequent misses.

The presence of multiple versions of the same line in the system also causes additional messages.

Specifically, when a processor requests a line, multiple versions of it may be provided, and the processor (or the directory) then selects what version to use.

Finally, the operation of speculative cache coherence protocols can also increase the traffic. The reason is that it is desirable to track dependences at a fine grain. To see why, recall how these protocols typically track dependences: they record which data are written and which data are exposed-read in each task (Section 2). This information is often encoded with a Write (W) and an Exposed-Read (R) bit per cached datum.

If this access information is kept per line (Figure 1-(a)), tasks that exhibit false sharing may appear to violate data dependences and, as a result, cause squashes [3]. For this reason, many SM proposals keep some access information at a finer grain, such as per word (Figure 1-(b)). Unfortunately, per-word dependence tracking induces higher traffic: a message (such as an invalidation) may need to be sent for each and every word of the line.

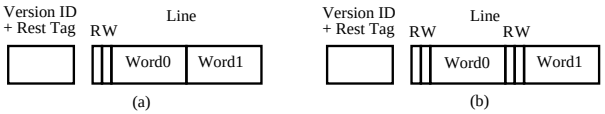


Figure 1: Keeping access information per line (a) or per word (b).

3.1.4 Additional Instructions

SM systems with compiler-generated tasks (such as ours) often execute more dynamic instructions than non-SM systems. There are two sources of these additional instructions: side-effects of breaking the code into tasks and, less importantly, SM-specific operations.

The majority of additional instructions result from two side-effects of task generation. First, conventional compiler optimizations are not very effective at optimizing code across task boundaries. Therefore, code quality is relatively lower. Secondly, in CMPs where processors communicate only through memory, the compiler often spills too many registers across task boundaries.

SM-specific operations are the other source. They include task spawn and commit instructions. The spawn instruction involves sending some state from one processor to another. In our implementation, this state is the program counter, the stack pointer, and a handful of other values. In other implementations, it may also involve executing a few instructions in the sender or receiver. Efficient, lazy implementations of task commit consist in sending the commit token from one processor to another [11] (Section 2). They do not involve any significant transfer of data or messages in the system: committed data are later transferred to memory on cache replacements. This is the approach that we use. In other implementations, commit may also involve executing a few instructions in both processors [5]. Overall, given the modest frequency of spawns and commits, their combined energy is very small.

3.2 Energy-Centric Optimizations

To reduce the energy consumed by these sources, we could use many performance-oriented SM optimizations proposed elsewhere. Examples are mechanisms to reduce the number of squashes [4, 16] or improvements to the speculative coherence protocol [15]. While these optimizations improve performance, they typically also reduce the energy consumed by a program.

In this paper, we are not interested in these optimizations. If they are effective, they should already be included in any SM design before addressing energy concerns. Instead, we are interested in "energy-centric" optimizations. These are optimizations that do not increase performance noticeably; in fact, they may slightly reduce it. However, they reduce energy significantly. We focus on these optimizations because they have been traditionally overlooked in performance-centric SM designs.

The energy-centric optimizations that we propose are shown in the second column of Table 1.

3.2.1 Task Squashing

We propose two optimizations to reduce energy consumption due to task squashing.

1. Stall a Task After Its Second Restart. When a task that has caused a data dependence violation and has been restarted already once causes a second violation, we propose to stall it for good. The task is not given a CPU again until it becomes non-speculative. This optimization is energy-centric: a performance-only approach would keep re-executing the task with the hope that one of the runs completes without violations.

Note that, when a task receives its first restart signal, we re-execute it immediately. We do this hoping to reuse the state in the branch predictor and caches. Often, the first data dependence violation is due to the passing of unexpected start-up information between parent and child and, after restart, no more violations will occur. A second violation may indicate the existence of too many true dependences to make speculative execution worthwhile. Consequently, we stall the task.

2. Energy-Aware Task Pruning by Profiling. Careful task pruning by a profiler pass attempts to allow only those squashes that are beneficial. Specifically, we propose an energy-centric profiler that attempts to keep the product $Energy \times Delay^2$ for the program low. Any task squash that increases the product is avoided because voltage-frequency scaling can (ideally) do better.

Our SM compiler generates a binary with tasks and spawn instructions to start-up tasks at run time (Figure 2-(a)). The binary is passed to a profiling pass, which executes it sequentially, using a profiling input (Section 5.2). The profiler estimates if a task squash will occur and, if so, the number of instructions squashed $I_{squashed}$ (Figure 2-(b)) and the final instruction overlap after re-execution $I_{overlap}$ (Figure 2-(c)). In addition, the profiler estimates the number of misses in the machine's L2 cache for the squashed instructions $M_{squashed}$. These misses will have a prefetching effect that will speed up the re-execution of *T2*.

The estimated execution time reduction is $I_{overlap} + T_0 \times M_{squashed}$, where T_0 is the estimated stall per L2 miss and it is assumed that each instruction takes one cycle to execute. The profiler

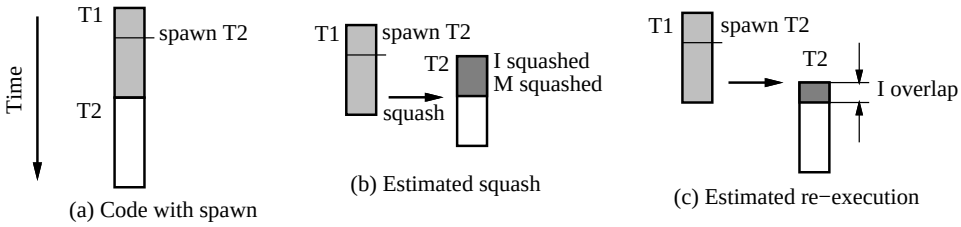


Figure 2: Estimating the benefits of a task squash.

also estimates the energy consumed by $I_{squashed}$. If the profiler estimates that allowing the spawn of *T2* degrades the $E \times D^2$ product of the program, it requests the removal of the spawn.

3.2.2 Storage and Logic for Data Versioning in the Memory Hierarchy

In SM, there are operations that require changing the tag state of *groups* of cache lines. For example, when a task is killed, its dirty cache lines are invalidated. Also, in eager-commit systems, when a task commits, its dirty cache lines are merged with main memory through ownership requests [15] or write backs [9]. Finally, in lazy-commit systems, when a cache has no free Local IDs (LID from Section 2) left, it needs to recycle one. This is done by selecting a long-committed task and writing back all its cache lines to memory. Then, that task's LID becomes free and can be re-assigned to a new local task.

Proposed SM schemes support these operations with energy-intensive actions — or expensive hardware. We want to avoid both.

For some operations, some existing schemes use a hardware finite state machine (FSM) that, in the background, repeatedly walks the tags in the cache. This is done for operations such as recycling LIDs. For example, in [12], a background FSM regularly does the following: it selects the LID of a committed task from the LID Table, walks the cache tags searching for lines from that task and writing them back, and finally frees up the LID. The FSM operates *eagerly*, using free cache cycles. Eager execution delivers the highest performance. Other existing schemes perform similar hardware walks of a group of (or all) the addresses in the cache, while stalling the processor to avoid causing hardware races in the cache. For example, this is the approach followed by [15] to commit the lines of a task: a special hardware module asks ownership for a group of cache lines whose address is stored in a buffer. Finally, for some operations, some existing schemes use "one-shot" hardware signals that change the tag state of a group of lines in a handful of cycles. For example, [3, 8, 15] do so to invalidate the dirty lines of a killed task. However, in multi-version caches, this operation may adversely affect the cycle time.

To save energy in these group operations, we propose two energy-centric optimizations.

1. Avoid Eagerly "Walking" the Cache Tags. To save energy, we want to perform all these group operations lazily in the background, especially avoiding any eager walk of the cache tags. Eager operation, even when there are free cycles, consumes energy that may not be fully justifiable. We only activate an eager background FSM in one case: to recycle LIDs when the cache is about

to run out of them. Specifically, we do it when there is only one free LID left. Overall, while this optimization may sometimes improve performance slightly, its main attraction is that it saves energy.

Our optimization relies on the table that contains translations from LIDs to global task IDs (LID Table from Section 2). Each entry in the LID Table is extended with summary use information for that LID: the number of lines that the corresponding task has in the cache, and whether the task is killed or committed. With this information, all the operations discussed above are performed in the background, lazily, and without address walking.

For example, consider a task kill or commit. When a task is killed or committed, its LID Table entry is updated by setting the Killed or the Committed bit, respectively (Figure 3-(a)). No tag walking is performed. Assume that, later, space is needed in a cache set that has no invalid line. As part of the (off-critical path) displacement algorithm, the LID Table is accessed for the lines in the cache set (Figure 3-(b)). For the entries that have the Killed bit set, the count of cached lines is decremented, and the corresponding lines in the cache are either chosen as the replacement victim or invalidated. Also, for the entries with the Committed bit set, the count is decremented, and the lines in the cache are written back to L2 to make room. If any one of these counters reaches zero, that LID is recycled. This continuous LID recycling practically eliminates the need for tag walking.

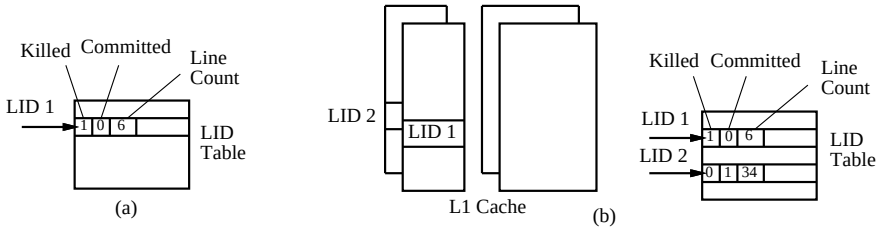


Figure 3: Using the LID Table on a task kill (a) and a cache line replacement (b).

2. Low-Energy Reuse of Cached Data on Task Restart. When a task is restarted after a violation, it should be able to reuse any clean lines remaining in the cache from its previous execution with minimal energy. This is relatively easy to do in existing schemes that do not restart a task until the hardware invalidates all the dirty cache lines of the task [3, 15]. Such schemes can simply give the same LID to restarted task, which will then trivially reuse the cached data.

Since our scheme restarts a task immediately, and only invalidates the task's dirty cache lines lazily, we cannot give the same ID to the task — the task would reuse invalid data. However, if we give the task a different LID, it is harder to reuse cached lines. Specifically, on a cache miss, the task needs to access the LID Table to see if any of the clean lines in the target set belong to a previous execution of the same task. Such lines will have a different LID but the same global task ID in the LID Table.

We propose an energy-centric optimization that sometimes eliminates this access to the LID

Table. The optimization consists in assigning to each task a combination of LID and *LID Offset (LIDOff)*. A task begins with a LIDOff set to zero; if it gets restarted, it keeps the same LID and increments its LIDOff. Moreover, cache tags include both LID and LIDOff. With this support, if an access finds that a cache line has the same LID and a lower LIDOff, it is a line from a previous execution of the same task. If the line is clean, the access is treated as a hit, and the line is *Promoted* by updating its LIDOff to the current value. The LID Table is not accessed. Avoiding the LID Table access affects performance little, as an out-of-order processor could hide the needed cycle(s). However, it saves energy.

4 Speculative Multithreaded CMP Architecture for Energy Efficiency

Based on the previous discussion, we now outline the architecture of our energy-efficient CMP with SM. We will use this architecture in our evaluation. While we have modeled in detail all aspects of the chip micro-architecture, we can only give an overview here. In the following, we describe the hardware structures and then the functionality.

4.1 Hardware Structures

We use a small-scale CMP with two (or four) modest-issue processors connected in a virtual ring. Each processor has a private, multi-versioned L1. The ring is also connected to a small, multi-versioned victim cache that holds lines overflowing from the L1s. Finally, there is an unmodified, shared L2 that only holds safe data (Figure 4-(a)). We chose a ring interconnect because it minimizes the number of hardware races in the speculative coherence protocol. At the same, it delivers high performance for the small number of processors considered. Also, we include a victim cache to avoid the much more expensive alternative of designing a multi-versioned L2. The combined space of the L1s plus the victim cache is practically always sufficient to hold the speculative state of all the running tasks; only rarely does a task get squashed due to lack of cache space. Figures 4-(b) to (d) show the extensions required by SM to processors, L1 caches, and victim cache. Each structure shows its fields in the form bit_count:field_name.

Each processor has an array of *TaskHolders*, which are hardware structures that hold some minimum state for the tasks that are currently loaded on the processor (Figure 4-(b)). Each TaskHolder contains the task's LID, LIDOff (incremented every time the task causes a violation, as per Section 3.2.2), a Stalled bit (set when the task causes a second violation and is forced to stall as per Section 3.2.1), Safe and Finished bits (set when the task receives the commit token and finishes execution, respectively), the task spawn address (PC), its stack pointer (SP), and a pointer to the next free TaskHolder. The TaskHolder does not store the register state, which is kept in the stack. A TaskHolder can be recycled when the owner task has committed and passed the commit token to its successor.

A copy of the LID and LIDOff for the task currently going through rename is kept in the *CurrentID* register of the load-store queue (Figure 4-(b)). Such register is used to tag loads and stores as they are inserted in the load-store queue. With this support, the processor can have

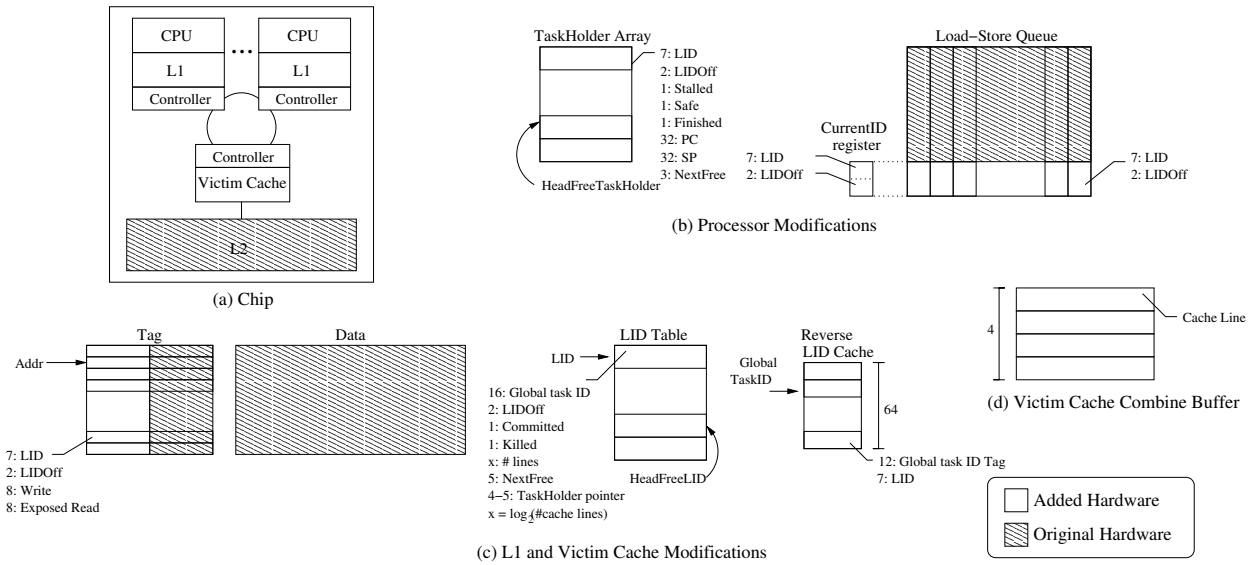


Figure 4: Proposed architecture for an energy-efficient speculative multithreaded CMP.

multiple in-flight tasks, and all accesses to the multi-versioned caches carry with them the correct LID and LIDOff.

In the L1s and the victim cache, each line tag is augmented with LID, LIDOff and, for each word in the line, one Write and one Exposed-Read bit to record accesses (Figure 4-(c)). As per Section 2, each cache keeps its own LID to global task ID translations in a LID Table (Figure 4-(c)). The LID Table is direct mapped and is indexed by a LID. Each entry has information for one LID: its corresponding global task ID, LIDOff, bits that indicate if the task is killed or committed, a counter of the number of lines in the cache belonging to that LID (Section 3.2.2), a global pointer to the corresponding TaskHolder, and a pointer to the next free entry. An entry can be recycled when its line counter is zero and the TaskHolder it points to has been recycled.

LIDs are local per cache. Since lines passed between caches include the global task ID, each cache needs a small *Reverse LID Cache* to translate from global task ID to own LID (Figure 4-(c)). If an access to the Reverse LID Cache misses, the LID Table is traversed, and a new entry is allocated in the Reverse LID Cache.

Finally, the victim cache has a *Combine Buffer* that is used when a safe line is about to be displaced to L2 (Figure 4-(d)). The buffer first requests from the caches all the safe versions of that line. As they arrive, the buffer combines them, so that each word in the line has the latest safe version on chip. Then, the line is committed to L2, and the other versions invalidated.

4.2 Functionality

To see how these structures are used, we now describe a task spawn, load hit and miss in L1, line displacement to L2, and task restart. Space limitations preclude giving more examples.

Task Spawn

When a processor executes a spawn instruction, it sends a small packet with the starting PC, SP, and global task ID to another processor. In the latter, the hardware allocates a new TaskHolder and initializes it as follows: LID is set to the value pointed to by HeadFreeLID (Figure 4-(c)); LIDOff, Stalled, Safe, and Finished are reset; and PC and SP are set to the values received in the message. The fields in the corresponding LID Table entry are also initialized. The global task ID is set with information from the message, the LIDOff, Committed, Killed, and line counter fields are reset, and the TaskHolder pointer is set to point to the TaskHolder. At this point, the task is ready to execute.

When the first instruction of a task goes through the rename stage, the LID and LIDOff from the TaskHolder are copied to the CurrentID register in the load-store queue. At any time, when an entry in the load-store queue is filled, it is also tagged with the CurrentID. As a result, when a load or store request is issued to L1, it carries with it the task's LID and LIDOff.

Load Hit/Miss in L1

If a load's address, LID, and LIDOff match one of the L1 tag entries, a hit is recorded and the data is returned immediately. If, instead, both address and LID match but LIDOff mismatches, the Write bits in the line are checked. If all of them are zero, the data is also returned as in a hit, and the line is promoted by setting the tag's LIDOff to the request's LIDOff (Section 3.2.2). This operation also clears the Exposed-Read bits except for the loaded word. Note that the LID Table is *not* accessed in either case.

In all other cases, a miss is recorded and the LID Table is accessed. We index the LID Table with the request's LID, obtain the corresponding global task ID, and include the latter in a request issued to the ring. Moreover, to decide which line to displace from L1, we also index the LID Table with the LIDs of the lines that are currently using the L1 set where space is needed. All these non-critical accesses proceed as fast as the number of read ports in the LID Table would allow. With the information retrieved from the LID Table, we can select the victim line — for example, one whose LID Table entry has the Killed bit set. If the victim line has to be displaced to another cache, the victim's LID Table entry provides the global task ID to include in the message. In all cases, the victim's LID Table entry is updated by decrementing its count of lines in L1.

A miss request deposited on the ring will reach all the other L1 caches in turn and the victim cache at the end. When a cache receives the request, it checks if it has a version of the line. It is possible that several lines match the address. The LIDs of these lines are used to index the local LID Table and retrieve the corresponding global task IDs to assess the relative order of the versions. The cache will assemble the lastest local version of the line that still precedes the requester task [11]. If any such line is generated, it is combined with the request and placed back on the ring.

This process is repeated by all caches and the victim cache which, all together, end up assembling the lastest version of the line on chip. After the victim cache completes its operation, the line is forwarded to the requester. If no matching line was found on chip, the victim cache initiates a read to L2.

Line Displacement

When a cache needs space, it can displace a committed or uncommitted line by dumping it on the ring. The other L1s and the victim cache will in turn attempt to absorb it. For a cache to absorb an incoming line, it needs to know the local LID that corresponds to the line's global task ID. Such LID is needed to insert the line in the receiving cache. To find this LID, the global task ID is used to index the Reverse LID Cache of the receiving cache. If a matching entry is found, it returns the LID. Otherwise, the LID Table is traversed to find out if a local translation exists. Note that this operation is not time critical. If no translation exists, a new one is created. The cache can now attempt to absorb the line. However, if all the entries in the target set are used, the cache makes room by shedding a committed line or a line more speculative than the incoming one. If no room can be found, the incoming line is rejected.

If, after the victim cache completes its operation there is still an uncommitted line that cannot be absorbed, the task that owns it is restarted. If the line that cannot be absorbed is safe, the victim cache uses the Combine Buffer to send it to L2 (Section 4.1).

Task Restart

When a cache detects a violation, the TaskHolder of the task that performed the stale read is examined. If the task is running, it is stopped. Its state is then re-started and its LIDOff in the TaskHolder is incremented. In addition, a message is broadcast on the ring to increment the LIDOff for the task in all the LID Tables. After that, if the task's LIDOff is 2, it means that this is its second restart. In this case, the Stalled bit in the TaskHolder is set and the task is stalled until it becomes safe. Otherwise, the task is allowed to re-execute.

5 Compilation Support

We have built a compiler for SM that automatically generates code out of sequential applications. Our compiler adds several passes to an experimental branch of gcc 3.5. The branch uses a static single assignment tree as the high-level intermediate representation [6]. With this approach, we leverage a complete compiler infrastructure, including a sophisticated control flow graph structure. In addition, the high-level intermediate representation of this branch is the right level for our work. If we were working at the source-code level, our transformations would likely be affected by unwanted compiler optimizations. Moreover, if we were working with a low-level representation such as RTL, we would have worse information, and it would be harder to perform pointer and dataflow analysis.

The resulting code quality, both when we enable and disable SM, is comparable to the MIPSPro SGI compiler for integer codes at the O3 optimization level. This is because, in addition to using a much improved gcc version, we also use SGI's source-to-source optimizer (copt from MIPSPro). The latter performs PRE, loop unrolling, inlining, and other optimizations.

To give a flavor of the compiler, we outline task generation and our profiling for energy efficiency.

5.1 Task Generation

Our compiler generates tasks out of loop iterations and the code that follows (i.e. the continuation of) subroutines. We handle recursivity seamlessly. Figure 5 shows how the compiler generates tasks. Figure 5-(a) shows the dynamic execution into and out of a loop; Figure 5-(c) shows the same idea for a subroutine. The compiler first marks the tasks, which are the loop iterations and the subroutine continuation. Then, it tries to place spawn statements for each of these tasks as early in the code as it can. A spawn is moved up in the code as long as the new position dominates the old one and both positions have execution equivalence. We do not move spawns up past statements that can cause data or control dependence violations. As examples, Figures 5-(b) and (d) show the resulting code for the loop and subroutine, respectively. In the figure, tasks on the right side are more speculative.

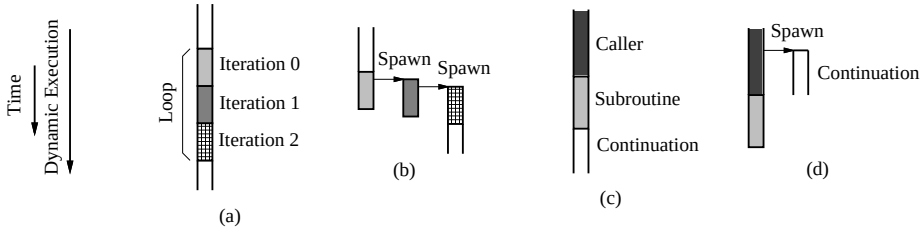


Figure 5: Generating tasks out of loop iterations and subroutine continuations.

Task generation algorithms try to maximize performance and minimize energy waste in multiple ways. For example, small subroutines are inlined rather than extracted. Loop iterations need to have a minimum size to be extracted as tasks. Finally, the compiler has a clean-up pass where it checks for tasks whose spawn was moved up the code no more than a handful of instructions. In this case, the spawn is removed, and the task not extracted.

5.2 Using Profiling for Energy Efficiency

The compilation process includes a simple profiler. The profiler takes the initial SM executable and identifies those task spawn points that should be eliminated because they are likely to induce harmful squashes according to our models. The profiler returns the list of such spawns to the compiler. Then, the compiler generates the final executable by removing those spawns and integrating the target tasks of those spawns with their statically predecessor tasks. On average, the profiler takes about two minutes to run.

The profiler executes the binaries sequentially, using the Train data set for SpecInt codes. As the profiler executes a task, it records the variables written. As it executes tasks that would be spawned earlier, it compares the addresses read against those written by predecessor tasks. With this, it can detect potential violations. The profiler also models a cache to estimate the number of misses in the real machine's L2, although no timing is modeled.

The profiler identifies those spawns that have a squash per commit rate higher than R_{squash}. For those spawns, it estimates $I_{squashed}$, $I_{overlap}$, and $M_{squashed}$ as in Figure 2. It also assumes an

average energy per instruction E_0 and an average stall time per miss T_0. The profiler can operate in performance or energy mode. In performance mode, it requests spawn removal if $I_{overlap} + T_0 \times M_{squashed}$ is less than a threshold T_{perf}. In energy mode, it requests spawn removal if subtracting $I_{overlap} + T_0 \times M_{squashed}$ from the program time and adding $I_{squashed} \times E_0$ to the program energy, increases the program's $E \times D^2$ product. The values for the thresholds and parameters used are listed in Section 6.

6 Evaluation Setup

To evaluate the energy efficiency of SM, we compare a SM chip with multiple narrow-issue processors to a non-SM chip with a single conventional, wide-issue processor. We use MIPS ISA execution-driven simulations with detailed models of out-of-order superscalar processors and memories, enhanced with models of dynamic and leakage energy from Wattch [1], Orion [20], and HotLeakage [23].

Architectures Evaluated

The SM CMP that we propose has 2 4-issue cores and the micro-architecture described in Section 4. Each core is similar to an up-to-date version of Alpha 21264. We call the chip *SM2-4i*. The non-SM chip has a single 8-issue superscalar processor with a conventional L1 and L2 on-chip cache hierarchy. The core is similar to Alpha 21464. We call the chip *Uni-8i*. Table 2 shows the parameters for these two chips.

While these two chips are different, we think that they provide reasonable design points to compare SM vs non-SM. We have scaled all the chip structures (ports, FU units, etc) according to the issue width of the core. Moreover, we try to favor *Uni-8i*. For example, all processors cycle at the same frequency and have a pipeline depth tuned for a 4-issue processor. This helps *Uni-8i*, since a real 8-issue processor would probably not cycle as fast and would have a more complex pipeline. Since the L1 caches in *SM2-4i* are multi-versioned, we increase their round-trip time from the processor one extra cycle relative to *Uni-8i*. Both processors have the same branch predictor because they have the same pipeline and branch misprediction penalty. Finally, all processors have an int and a fp functional unit cluster. Since we use integer codes for the evaluation, the fp cluster is clock-gated almost all the time.

In our evaluation, we give all execution speedups relative to a non-SM chip with a single 4-issue superscalar (*Uni-4i*). This chip has a single core like one in *SM2-4i* and a conventional memory hierarchy like *Uni-8i* (except that the L1 cache only has 2 ports).

For completeness, we also evaluate a SM chip like *SM2-4i* but with 4 cores. We call it *SM4-4i*.

Energy Considerations

We estimate the energy consumed in all chip structures, including processor, memory hierarchy and on-chip interconnect. For the dynamic energy, we use models from Wattch [1] and Orion [20]. We apply clock gating as in Wattch, assuming that a clock-gated structure consumes at most 10%

2-processor 4-issue SM CMP (*SM2-4i*)					8-issue superscalar (*Uni-8i*)			
Processor					Processor			
Frequency: **5.0 GHz**				FE/IS/RE width: 8/**4**/6	Frequency: **5.0 GHz**			FE/IS/RE width: 16/**8**/12
Technology: 90 nm				I-window/ROB size: 96/192	Technology: 90 nm			I-window/ROB size: 160/360
Branch penalty: 17 cyc (min)				Int/FP registers: 128/112	Branch penalty: 17 cyc (min)			Int/FP registers: 224/160
RAS: 32 entries				Ld/St units: 2/2	RAS: 32 entries			Ld/St units: 4/4
BTB: 2K entries, 2-way assoc.				Int/FP units: 3/2	BTB: 2K entries, 2-way assoc.			Int/FP units: 7/4
Branch predictor:				Ld/St queue entries: 64/48	Branch predictor:			Ld/St queue entries: 128/128
Hybrid with speculative update				TaskHolders/processor: 8	Hybrid with speculative update			
Bimodal size: 16K entries				TaskHolder acc time: 1 cyc	Bimodal size: 16K entries			
Gshare-11 size: 16K entries				TaskHolder acc energy: 0.25nJ	Gshare-11 size: 16K entries			
Cache	L1	VC	L2	Bus & Memory	Cache	L1	L2	Bus & Memory
Size:	8KB	4KB	1MB	FSB frequency: 533MHz	Size:	8KB	1MB	FSB frequency: 533MHz
RT:	**2 cyc**	5 cyc	10 cyc	FSB width: 128bit	RT:	**1 cyc**	8 cyc	FSB width: 128bit
Assoc:	4-way	4-way	8-way	Memory: DDR-2	Assoc:	4-way	8-way	Memory: DDR-2
Line size:	32B	32B	64B	DRAM bandwidth: 8.528GB/s	Line size:	32B	64B	DRAM bandwidth: 8.528GB/s
Ports:	2	1	1	Memory RT: 81.6ns	Ports:	4	1	Memory RT: 81.6ns
MSHRs:	128	96	96		MSHRs:	128	96	
LID Table:								
entries:	128	128						
ports:	2	2						
acc time:	1 cyc	1 cyc						
acc energy:	0.25nJ	0.25nJ						
RLC:								
size:	64	64						
assoc:	4	4						
ports:	1	1						
acc time:	1 cyc	1 cyc						
acc energy:	0.25nJ	0.25nJ						

Profiling parameters:			
R_{squash}: 0.55	E_0: 8pJ	T_0: 390 cyc	T_{perf}: 100 cyc

Table 2: 2-processor 4-issue SM CMP (*SM2-4i*) and 8-issue superscalar (*Uni-8i*) chip architectures modeled. In the table, FE, IS, RE, MSHR, RAS, FSB, RT, RLC, and VC stand for fetch, issue, retirement, Miss Status Handling Register, Return Address Stack, Front-Side Bus, minimum Round-Trip time from the processor, Reverse LID Cache, and Victim Cache, respectively. Cycle counts refer to processor cycles.

its original energy. We extend the Wattch model to consider the deeper pipelines we are modeling. We also have an H-tree model for clock distribution, which computes clock energy based on area. The leakage energy is estimated with HotLeakage [23]. In our analysis, we find that leakage energy is less than 5% of the dynamic energy. This figure was verified with the authors of HotLeakage [13]. Consequently, we neglect leakage energy in our calculations. Some energy parameters for our chips are shown in Table 2.

Applications Evaluated

Our architectures run the SpecInt 2000 applications with the Ref data set. The exceptions are *eon*, *gcc*, *perlbmk* (where the compiler fails), and *vortex* (where the simulator fails). *Uni-4i* and *Uni-8i* run SpecInt 2000 binaries compiled with our SM pass disabled. The code quality is comparable to the MIPSPro SGI compiler for integer codes at O3 level. *SM2-4i* and *SM4-4i* run binaries compiled with our SM pass enabled.

SM and non-SM binaries are very different. To compare architectures running different binaries, we cannot compare the execution of a fixed number of instructions. Instead, we insert "simulation markers" in the code and simulate for a given number of markers. After skipping the initialization (typically 1-6 billion instructions), we execute up to a certain number of markers so that *Uni-8i* graduates more than 500 million instructions.

7 Evaluation

7.1 Qualitative Insights

A SM chip such as *SM2-4i* suffers from all the SM-specific sources of energy consumption discussed in Section 3.1. These sources are repeated in Column 1 of Table 3. However, the non-SM chip (*Uni-8i*) also has additional sources of energy resulting from having a wider core. These sources are shown in Column 2 of Table 3.

Additional Sources of Energy in *SM2-4i*	Additional Sources of Energy in *Uni-8i*
Task squashing	More ports in structures
Data versioning in the memory hierarchy	Larger structures
More traffic in the memory subsystem	Larger clock contribution
More graduated instructions (committed tasks only)	More instructions squashed by branch mispredictions

Table 3: Comparing the *additional* sources of energy consumption.

One source of energy consumption in *Uni-8i* is its higher number of ports in many structures. Relative to a core in *SM2-4i*, the *Uni-8i* core doubles the number of ports in the register file, instruction window, reorder buffer, data cache, and load-store queue. Doubling the number of ports in a structure roughly doubles the energy spent per access to the structure [21]. In addition, all these structures except the cache are larger in *Uni-8i* than in a *SM2-4i* core. This also increases the consumption per access. Moreover, *Uni-8i* also spends more energy in the clock network. One important reason for this is that each core in *SM2-4i* has its own clock domain, which simplifies

clock distribution and reduces its energy cost. Another reason is that the *Uni-8i* chip is a bit larger than *SM2-4i*, since it would take about 3 cores in the SM chip to equal the size of *Uni-8i*. Finally, while *SM2-4i* ends up executing more graduated instructions (counting committed tasks only), *Uni-8i* fetches more *non-graduated* instructions. The reason is that, with the same pipeline and branch predictor, the wider processor squashes more instructions at every branch misprediction.

7.2 Overall Comparison

We start by comparing the performance and the average power consumption of *SM2-4i* and *Uni-8i* running the applications. Figure 6-(a) shows the speedups of these chips (and *SM4-4i*) relative to the execution on *Uni-4i*; Figure 6-(b) shows the average power consumption of all the chips during execution.

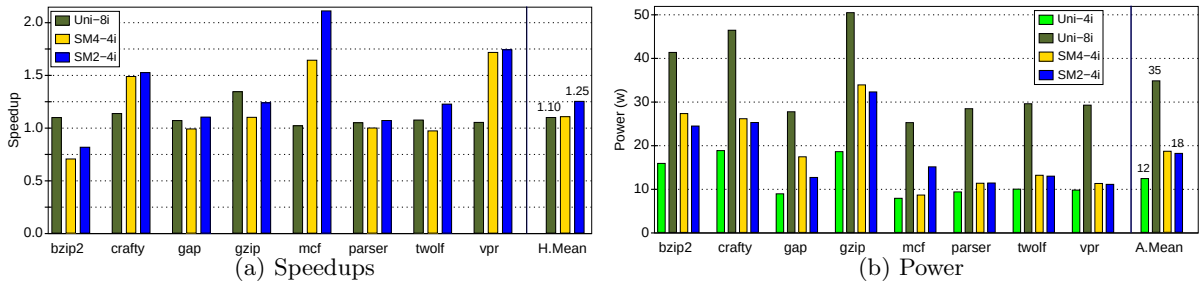


Figure 6: Execution speedup relative to *Uni-4i* (a) and average power consumption (b) for different chip organizations. *SM2-4i* and *SM4-4i* are SM CMPs, while *Uni-4i* and *Uni-8i* are conventional superscalar chips.

Figure 6-(a) shows that *SM2-4i* is faster than *Uni-8i* for the majority of applications. This shows that the SM compiler successfully extracts tasks from these irregular codes. *Uni-8i* speeds-up up the codes, but it is not as effective as *SM2-4i*. The only application where *Uni-8i* is significantly faster is bzip2. The reason is that bzip2 has small tasks (e.g. 65 instructions), which increases the overheads in *SM2-4i*. On average, *SM2-4i* is 13% faster than *Uni-8i*.

On the other hand, from Figure 6-(b), we see that the average power consumed by *SM2-4i* is much lower than in *Uni-8i*. On average, it is about half. While *Uni-8i* clock gates unused structures, the effects shown in Table 3 boost the power consumed by *Uni-8i*. Overall, if we consider both performance and power, we see that *SM2-4i* represents a significantly better design point than *Uni-8i*.

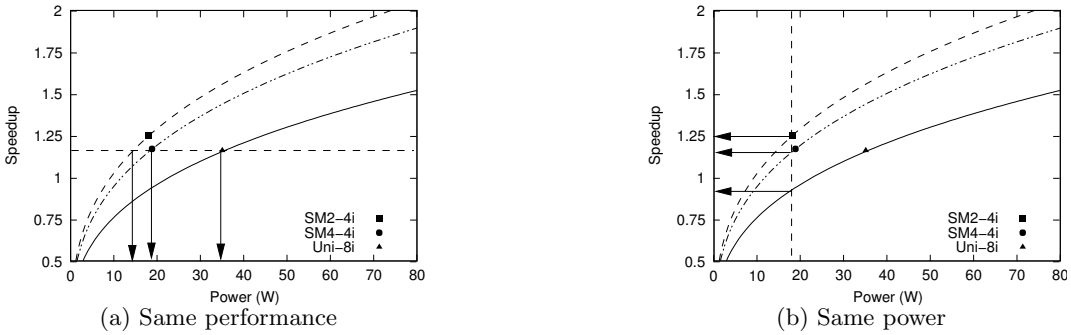
Figure 6 also shows that *SM4-4i* is not as energy-effective as *SM2-4i*. Placing four cores on a SM chip reduces the speedups while maintaining the power consumed. The speedups decrease largely because of memory system effects: more caches means lower locality and higher communication overhead for limited parallelism. On the other hand, for more parallel applications such as SpecFP, *SM4-4i* is likely to be better.

Finally, *Uni-4i* consumes the least power; however, *SM2-4i* is 25% faster for these codes.

To gain more insight, we can compare *SM2-4i* and *Uni-8i* at a fixed average power and at a

fixed performance. For this, we *analytically* apply ideal voltage-frequency scaling to the chips. We assume that speedup is linearly proportional to frequency and average power is proportional to the cube of frequency. Then, we can represent a curve that relates speedup (Sp) and average power (P) as $Sp = k * \sqrt[3]{P}$.

Figure 7 shows this curve for *SM2-4i*, *Uni-8i*, and *SM2-4i*. The curves are replicated in Figure 7-(a) and (b). For example, the curve for *SM2-4i* shows the different speedup-power working points for the chip as we vary the frequency. In each curve, we show the actual working point of the chip that was measured in Figure 6 for the average of the SpectInt applications.



(a) Same performance (b) Same power

Figure 7: Ideal relation between speedup and average power for the different chips considered.

In Figure 7-(a), we set the performance to that measured for *Uni-8i*. We then scale down the frequency for *SM2-4i* and *SM4-4i*. We see that, at the same performance, *SM2-4i* (ideally) consumes approximately a third of *Uni-8i*'s average power. Similarly, *SM4-4i* (ideally) consumes approximately half of *Uni-8i*'s power.

In Figure 7-(b), we set the power to the one measured for *SM2-4i*. We then scale down the frequency of *Uni-8i* and *SM4-4i*. We see that, at the same average power, *SM2-4i* is (ideally) about 60% faster than *Uni-8i*, and *SM4-4i* is about 40% faster than *Uni-8i*. Overall, these analyses provide insight on the energy-effectiveness of *SM2-4i*.

7.3 Understanding the Additional Energy Consumption in SM

We now assess the additional sources of energy consumption from Table 3, starting with those in SM. Table 4 gives measurements from the execution of the applications on *SM2-4i* and *Uni-8i*. Column 2 shows the fraction of the energy consumed in *SM2-4i* that is spent in squashed tasks. We see that some applications such as mcf spend up to 21% of their energy in squashed tasks, while others like twolf or vpr spend no energy. On average, squashes are responsible for a significant 7.3% of the energy in *SM2-4i*.

Column 3 in Table 4 shows the fraction of the *SM2-4i* energy consumed by the logic and storage for data versioning in the memory hierarchy. For this category, we add up the contribution of all the white structures in Figures 4-(b), (c), and (d), plus the victim cache, and various logic associated with data versioning. From the table, we see that this energy is also significant. On average, it accounts for 7.8%. Section 7.5 further analyzes the energy consumed in *SM2-4i*'s memory system.

App	*SM2-4i*			*Uni-8i*
	Energy in Squashed Tasks (%)	Energy in Multi-version Structs (%)	Instr. Graduated *SM2-4i*/*Uni-8i* (Committed Tasks Only)	Instr. Squashed by BR Mispred. *Uni-8i*/*SM2-4i* (Committed Tasks Only)
bzip2	17.56	11.58	1.12	1.77
crafty	0.12	5.90	1.09	1.26
gap	14.92	2.27	1.08	0.82
gzip	4.32	9.76	1.24	1.08
mcf	20.57	8.22	1.82	0.76
parser	0.63	7.39	1.21	1.07
twolf	0.00	10.11	1.29	1.02
vpr	0.00	6.95	1.08	1.26
Avg	7.26	7.77	1.24	1.06

Table 4: Energy-related parameters in the execution of the applications on *SM2-4i* and *Uni-8i*.

We do not provide a breakdown of the energy in SM due to additional traffic because it is hard to accurately assess it. Many of the additional messages simply result from parallelizing the code; they have nothing to do with SM. However, we give some insights in Section 7.5.

Finally, Column 4 in Table 4 gives insight on the last source of SM-specific energy shown in Table 3, namely more graduated instructions. The column shows the number of graduated instructions in *SM2-4i* over those in *Uni-8i*. We only include committed tasks. If *SM2-4i* graduates more instructions, it is largely because its code is less efficient than *Uni-8i*'s. On average, *SM2-4i* graduates 24% more instructions.

7.4 Understanding the Additional Energy Consumption in non-SM Chips

To fairly assess the energy impact of SM, we also need to evaluate the additional sources of energy consumption in the non-SM chip (*Uni-8i*). These sources, shown in Table 3, are avoided in SM by virtue of using simpler cores.

One of these sources is the more numerous instructions squashed by branch mispredictions. The last column in Table 4 shows the number of such instructions in *Uni-8i* over that in *SM2-4i*. To measure this effect correctly, we again ignore all the *SM2-4i* instructions executed in squashed tasks. From the table, we see that *Uni-8i*'s branches squash on average 6% more instructions than *SM2-4i*'s. While this is not a negligible number, we had expected a higher number, given the wider *Uni-8i* pipeline. The reason for this low number is that, since *SM2-4i* has two processors, its branch predictors do not get trained as well as *Uni-8i*'s.

To evaluate the other additional sources of energy in non-SM chips, Figure 8 shows a breakdown of the total energy consumed during the execution of the applications. The figure shows bars for all the chips and normalizes them to *Uni-4i*. The bars are broken into energy spent in fetch, issue, execution, clock, and memory system.

Comparing *Uni-8i* and *SM2-4i*, we see the effect of the three remaining additional sources of energy in non-SM chips. One of the sources is higher clock energy (Table 3). As shown in Figure 8,

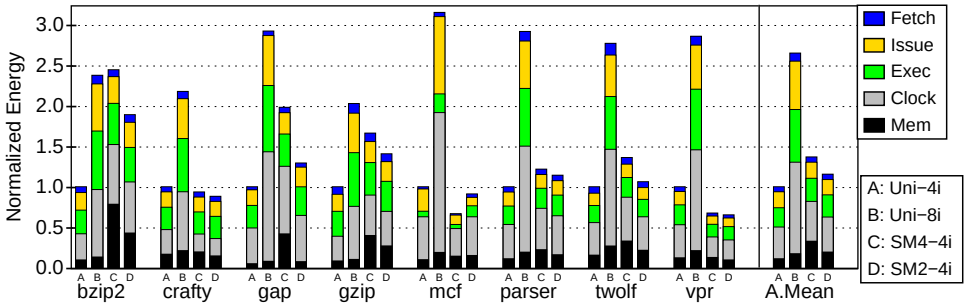


Figure 8: Breakdown of the energy consumed during the execution of the applications. The bars are normalized to the energy consumed by *Uni-4i*.

clock is the dominant source of energy consumption in *Uni-8i*. Its contribution is much higher than in *SM2-4i*.

The absolute and relative contributions of issue and execution are higher in *Uni-8i* than in *SM2-4i*. These categories contain many of the structures that increase in size or have more ports.

Finally, note that the absolute and relative contribution of the memory system to the energy consumption is higher in *SM2-4i* than in *Uni-8i*. To understand the behavior of the memory system in *SM2-4i*, we now examine its energy consumption in detail.

7.5 Analysis of the Energy Consumption in a SM Memory Hierarchy

Figure 9 shows a breakdown of the energy consumed in the memory hierarchy for the different chips. For each application, the bars are normalized to *Uni-4i*. The energy is broken into the contribution of the most significant structures in the memory hierarchy. L1, L2, victim cache, and ring are self-explanatory. LSQ includes the additional fields in the load-store queue and the processor shown in white in Figure 4-(b). LID includes the LID Tables and the Reverse LID Caches (Figure 4-(c)). Note that *Uni-8i* and *Uni-4i* only have L1 and L2.

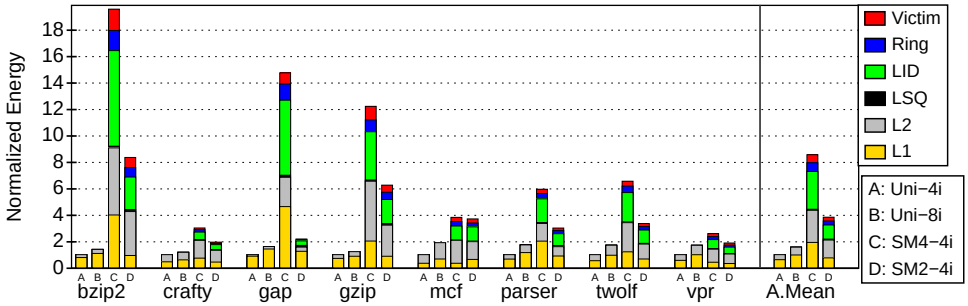


Figure 9: Breakdown of the energy consumed in the memory hierarchy for the different chips.

Figure 9 shows that the SM chips consume more energy in L1 and L2 than the non-SM chips. Recall that the L2 is the same in all the chips. Consequently, the higher energy consumption in L2 is due to the higher traffic in *SM2-4i* and *SM4-4i*. Some of this traffic is due to parallelism, while some other is due to SM transactions. Overall, caches are accessed more frequently under

SM, especially in *SM4-4i*.

The bars show a large contribution for LID. This is the core of the multi-version support in the memory hierarchy. Consequently, we conclude that it is important that the multi-version support in the memory hierarchy be designed for energy-efficiency. On the other hand, the contribution of the victim cache and load-store queue fields is small.

7.6 Impact of Energy-Centric Optimizations

Finally, we assess the impact of the energy-centric optimizations proposed in Section 3.2. These optimizations are included in *SM2-4i*. Figure 10 assesses the energy consumption of *SM2-4i* if we remove one of these optimizations at a time. The figure compares the original system (*SM2-4i*) to one without stalling a task after its second restart (*NoSyncOnRestart*), one without energy-aware task pruning by profiling (*NoEProf*), or one without the two optimizations of Section 3.2.2: no eager walking of the cache tags and low-energy reuse of cached data on task restart (*WalkTags*). In the figure, the bars are normalized to *SM2-4i*. Unless otherwise indicated, the performance is unaffected.

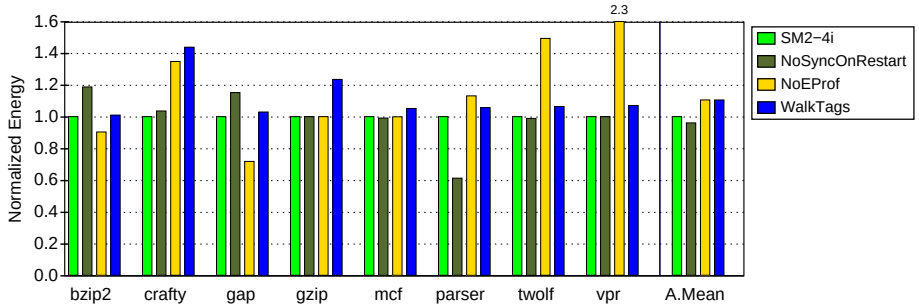


Figure 10: Effect of removing energy-centric optimizations. Unless otherwise indicated, the performance is unaffected.

Eliminating the stall on second restart typically increases the energy consumption or does not change it. However, there is the unusual case of parser, where repeatedly restarting one task ends up speeding up the program significantly. As a result, it saves energy. This case is unusual. Due to this single application, removing this optimization even saves a bit of energy on average.

Removing the task pruning optimization generally increases the energy consumption. In two cases, it has the opposite effect. On average, removing the optimization increases the energy by 13%. Finally, removing the tag walking optimizations also increases the energy consumed by an average of 13%. The impact of this optimization depends on application characteristics. If there are many small tasks or squashes are frequent, not having this optimization hurts both energy and performance. For example, some of our applications would run 1-2% slower.

Overall, it is difficult for optimizations to save a lot of the chip energy on average. However, two of the ones that we propose have a sizable impact for the SpecInt applications.

8 Conclusions

This paper challenges the commonly-held view that SM consumes excessive energy. We showed that it is possible to design a Chip Multiprocessor (CMP) with Speculative Multithreading (SM) that, for the same performance, consumes less power than a wide-issue superscalar. We demonstrated it with a new energy-efficient CMP micro-architecture. In addition, we identified sources of energy consumption in SM, which are the wasted work of squashed tasks, storage and logic in the memory hierarchy to support data versioning, additional traffic in the memory subsystem, and additional instructions. Finally, we proposed energy-centric optimizations that mitigate some of these sources.

In our experiments with the SpecInt 2000 benchmarks, we show that a CMP with 2 4-issue cores delivers a speedup of 1.08 over an 8-issue superscalar while consuming only 54% of its power. Alternatively, for the same average power in both chips, the SM CMP is 1.6 times faster than the superscalar on average.

We hope that this work helps propel SM into mainstream microprocessors. CMPs are attractive platforms because they are more energy-efficient, more scalable, and have lower complexity than conventional wide-issue superscalars. Moreover, they have an advantage for explicitly-parallel codes. In this paper, we showed that they can also speed up SpecInt-class applications more than conventional superscalars for a similar energy budget.

References

[1] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a Framework for Architectural-Level Power Analysis and Optimizations. In *Proc. 27th Annual Intl. Symp. on Computer Architecture*, pages 83–94, June 2000.

[2] M. Chen and K. Olukotun. The Jrpm System for Dynamically Parallelizing Java Programs. In *Proc. 30th Intl. Symp. on Computer Architecture*, June 2003.

[3] M. Cintra, J. F. Martínez, and J. Torrellas. Architectural Support for Scalable Speculative Parallelization in Shared-Memory Multiprocessors. In *Proc. 27th Annual Intl. Symp. on Computer Architecture*, pages 13–24, June 2000.

[4] M. Cintra and J. Torrellas. Eliminating squashes through learning cross-thread violations in speculative parallelization for multiprocessors. In *Proc. 8th High-Performance Computer Architecture Conf.*, Feb. 2002.

[5] M. J. Garzarán et al. Tradeoffs in Buffering Memory State for Thread-Level Speculation in Multiprocessors. In *Proc. 9th High-Performance Computer Architecture Conf.*, pages 191–202, Feb. 2003.

[6] SSA for trees - GNU project. URL, May 2003. "http://www.gccsummit.org/2003/view_abstract.php?talk=2".

[7] S. Gopal, T. Vijaykumar, J. Smith, and G. Sohi. Speculative Versioning Cache. In *Proc. 4th Intl. Symp. on High-Performance Computer Architecture*, pages 195–205, February 1998.

[8] L. Hammond, M. Willey, and K. Olukotun. Data Speculation Support for a Chip Multiprocessor. In *8th Intl. Conf. on Arch. Support for Prog. Lang. and OS*, pages 58–69, October 1998.

[9] V. Krishnan and J. Torrellas. A Chip-Multiprocessor Architecture with Speculative Multithreading. *IEEE Trans. on Computers*, pages 866–880, September 1999.

[10] P. Marcuello and A. Gonzalez. Clustered Speculative Multithreaded Processors. In *Proc. 1999 Intl. Conf. on Supercomputing*, pages 365–372, June 1999.

[11] M. Prvulovic, M. J. Garzarán, L. Rauchwerger, and J. Torrellas. Removing architectural bottlenecks to the scalability of speculative parallelization. In *Proc. 28th Intl. Symp. on Computer Architecture (ISCA'01)*, pages 204–215, June 2001.

[12] M. Prvulovic and J. Torrellas. ReEnact: Using Thread-Level Speculation Mechanisms to Debug Data Races in Multithreaded Codes. In *Proc. 30th Intl. Symp. on Computer Architecture*, pages 110 – 121, June 2003.

[13] K. Skadron. Personal communication. October 2003.

[14] G. Sohi, S. Breach, and T. Vijayakumar. Multiscalar Processors. In *22nd Intl. Symp. on Computer Architecture*, pages 414–425, June 1995.

[15] J. Steffan, C. Colohan, A. Zhai, and T. Mowry. A Scalable Approach to Thread-Level Speculation. In *Proc. 27th Annual Intl. Symp. on Computer Architecture*, pages 1–12, June 2000.

[16] J. Steffan, C. Colohan, A. Zhai, and T. Mowry. Improving Value Communication for Thread-Level Speculation. In *Proc. 8th High-Performance Computer Architecture Conf.*, February 2002.

[17] J. Steffan, C. B. Colohan, and T. C. Mowry. Architectural Support for Thread-Level Data Speculation. Tech. Rep., CMU-CS-97-188, Carnegie Mellon University, November 1997.

[18] M. Tremblay. MAJC: Microprocessor Architecture for Java Computing. Hot Chips, August 1999.

[19] J. Tsai et al. The Superthreaded Processor Architecture. *IEEE Trans. on Computers*, 48(9):881–902, September 1999.

[20] H. S. Wang, X. P. Zhu, L. S. Peh, and S. Malik. Orion: A Power-Performance Simulator for Interconnection Networks. In *Proc. 35th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-35)*, 2002.

[21] S. Wilton and N. Jouppi. CACTI: An Enhanced Cache Access and Cycle Time Model. *IEEE Journal on Solid-State Circuits*, 31(5):677–688, May 1996.

[22] A. Zhai, C. Colohan, J. Steffan, and T. Mowry. Compiler Optimization of Scalar Value Communication Between Speculative Threads. In *ASPLOS X Proceedings*, San Jose, CA, October 2002.

[23] Y. Zhang et al. Hotleakage: A temperature-aware model of subthreshold and gate leakage for architects. Tech. Rep. CS-2003-05, Univ. of Virginia Dept. of Computer Science, March 2003.

[24] C. Zilles and G. Sohi. Master/Slave Speculative Parallelization. In *Proc. 35th Annual Intl. Symp. on Microarchitecture*, pages 65–77, November 2002.

TLS Chip Multiprocessors: Micro-Architectural Mechanisms for Fast Tasking with Out-of-Order Spawn

Draft paper submitted for publication. November 6, 2003.

Please keep confidential

Jose Renau, James Tuck, Wei Liu, Luis Ceze, Karin Strauss, Josep Torrellas
University of Illinois

Abstract

Chip Multiprocessors (CMP) are flexible, high-frequency platforms on which to support Thread-Level Speculation (TLS). However, for TLS to deliver on its promise, CMPs must exploit multiple sources of speculative task-level parallelism, including any nesting levels of both subroutines and loop iterations. Unfortunately, these environments are hard to support in decentralized CMP hardware: since tasks are spawned out-of-order and unpredictably, maintaining key TLS basics such as task ordering and efficient resource allocation is challenging.

This paper is the first one to propose micro-architectural mechanisms that, taken together, fundamentally enable fast TLS with out-of-order spawn in a CMP. These simple mechanisms are: Splitting Timestamp Intervals, the Immediate Successor List, and Dynamic Task Merging. To evaluate them, we develop a TLS compiler with out-of-order spawn. With our mechanisms, a TLS CMP with 2 4-issue processors increases the average speedup of full SpecInt 2000 applications from 1.15 (no out-of-order spawn) to 1.25 (with out-of-order spawn). Moreover, the resulting CMP outperforms a very aggressive 8-issue superscalar. Specifically, with the same clock frequency, the CMP delivers an average speedup of 1.14 over the 8-issue processor.

1 Introduction

Chip Multiprocessors (CMP) with Thread-Level Speculation (TLS) are being put forward as flexible, high-frequency engines to extract the next level of parallelism from hard-to-analyze programs (e.g. [7, 8, 9, 11, 15, 16, 17, 18, 24]). In these architectures, irregular sequential codes are divided into tasks that are executed in parallel, optimistically assuming that sequential semantics will not be violated. As the tasks run, the architecture tracks their control flow and data accesses. If a cross-task dependence is violated, the offending tasks are destroyed (*squashed*). Then, a repair action is initiated and the offending tasks are re-executed.

While these architectures have shown good potential, often thanks to sophisticated compiler support [2, 5, 10, 19, 20, 23], the speedups obtained for non-numerical applications have typically been only modest. Part of the reason is that most designs have typically focused (often implicitly) on limited types of task structures: iterations from a single loop level (e.g. [4, 9, 23]), the code that follows (i.e. the continuation of) calls to subroutines that do not spawn other tasks (e.g. [3]), or

some execution paths out of the current task (e.g. [20]). In the cases mentioned, tasks are spawned *in order*, namely in the same order as they would in sequential execution. While exploiting only these task structures may simplify the CMP hardware, it cripples TLS potential.

High-level performance evaluation studies have pointed out that there is a sizable amount of other parallelism available [12, 13, 21, 22]. One can execute in parallel all subroutines and their continuations irrespective of their nesting, and iterations from multiple loop levels in a nest. If this additional parallelism is also leveraged, the speedups are predicted to be significantly higher.

In practice, exploiting these additional sources of parallelism requires supporting *out-of-order* task spawning. For example, consider nested subroutines. When a task finds a subroutine call, it spawns a more speculative task to execute the continuation, and proceeds to execute the subroutine. The same task can then find other subroutine calls, therefore spawning speculative tasks that are less speculative than the one spawned first. The same occurs for nested loops, and for combinations of loop and subroutine nesting.

With out-of-order spawning, the application offers unpredictable shapes of parallelism that are hard to manage at run time. The resulting TLS environment is challenging. Specifically, how do we manage task ordering, which is required to identify violations and to ensure commit and squash order? How do we balance resource allocation between highly speculative tasks that have been running for a long time, and just-spawned, less speculative tasks? To address these challenges with high speed in a CMP, we need special micro-architecture.

Unfortunately, no previous work has outlined such CMP micro-architecture. Hammond *et al.* [8] have proposed a TLS CMP system that supports out-of-order spawning with both subroutine and loop-iteration tasks of any nesting level. However, to control key parts of the speculation machinery, they use a co-processor running software handlers (Section 8). They conclude that their scheme has too much control software overhead to support subroutine parallelism.

Other work on tasking with out-of-order spawn has only focused on high-level performance evaluation [12, 13, 21, 22], often simulating ideal architectures. It has not described any micro-architecture design. The one design that presents some micro-architecture for out-of-order spawn is DMT [1], which is based on a single centralized, multithreaded CPU (Section 8). Such solution is not viable for the decentralized architecture of a CMP, which requires completely different support.

This paper is the first one to propose micro-architectural mechanisms that, taken together, fundamentally enable *high-speed* tasking with out-of-order spawn in a TLS CMP. These simple mechanisms support correct and efficient task ordering and resource allocation. Task ordering is enabled with *Splitting Timestamp Intervals* for low-overhead order management, and the *Immediate Successor List* for efficient task commit and squash. Efficient resource allocation is enabled with *Dynamic Task Merging*, which directs speculative parallelism to the most beneficial sections of the code.

To test our micro-architecture, we develop a gcc-based TLS compiler for out-of-order spawn. We show that a simulated TLS CMP with 2 4-issue processors increases the average speedup of

full SpecInt 2000 applications from 1.15 (no out-of-order spawn) to 1.25 (with out-of-order spawn). Moreover, the resulting CMP outperforms an 8-issue superscalar: with the *same clock frequency*, the CMP delivers an average speedup of 1.14 over the 8-issue processor. Overall, our micro-architectural mechanisms are very effective at boosting TLS speedups.

This paper is organized as follows: Section 2 introduces out-of-order spawning; Sections 3 and 4 present our micro-architectural design and implementation; Section 5 describes the compilation infrastructure; Sections 6 and 7 present our evaluation methodology and the evaluation; and Section 8 discusses related work.

2 Speculative Tasking with Out-of-Order Spawn

In most of the proposed TLS systems, tasks are formed with iterations from a single loop level (e.g. [4, 9, 23]), the code that follows (i.e. the continuation of) calls to subroutines that do not spawn other tasks (e.g. [3]), or some execution paths out of the current task (e.g. [20]). In these proposals, an individual task can at most spawn one correct task in its lifetime. A correct task is one that is in the sequential execution path of the program. As a result, tasks are spawned *in order*, namely in the same order as they would in sequential execution.

Figures 1-(a) and (b) show examples. Figure 1-(a) shows the task tree when parallelizing a loop. Each task spawns the next iteration. In the figure, the leftmost task is safe (or non-speculative); the more a task is to the right, the more speculative it is. Figure 1-(b) shows the tree when a task finds a leaf subroutine. The original task continues execution into the subroutine, while a more speculative task is spawned to execute the continuation.

There is broad consensus that, for TLS to deliver on its promised speedups, it has to exploit more parallelism. Several high-level performance evaluation studies [12, 13, 21, 22], typically simulating simplified architectures, have pointed to the need to additionally support subroutines from any nesting level and iterations from multiple loop levels in a nest.

Figures 1-(c) and (d) show the two cases. In Figure 1-(c), the safe task first spawns a task for the continuation of subroutine *S1*. Then, it executes the beginning of *S1*, spawns a new task for the continuation of *S2*, and executes *S2* until its end. In Figure 1-(d), the safe task executes outer iteration 0. As it executes, it spawns outer iteration 1, enters the inner loop to execute inner iteration 0, and spawns inner iteration 1. When it completes inner iteration 0, it ends.

With these task choices, an individual task can spawn multiple correct tasks. If so, correct tasks are spawned in strict reverse order compared to sequential execution. For example, in Figures 1-(c) and (d), the safe task spawns two correct tasks, and does so out of order. Figure 1-(e) is a more complex example: the time-line for task creation proceeds from top to bottom (*1-2-3-4-5-6-7*), while sequential order is from left to right (*1-6-7-4-3-5-2*).

In the rest of the paper, to discuss out-of-order spawning, we give examples of tasks built out of any-nesting subroutines and loop iterations, as they are an obvious source of TLS parallelism. Our analysis also applies to any other task structure that maintains two conventions. First, if a

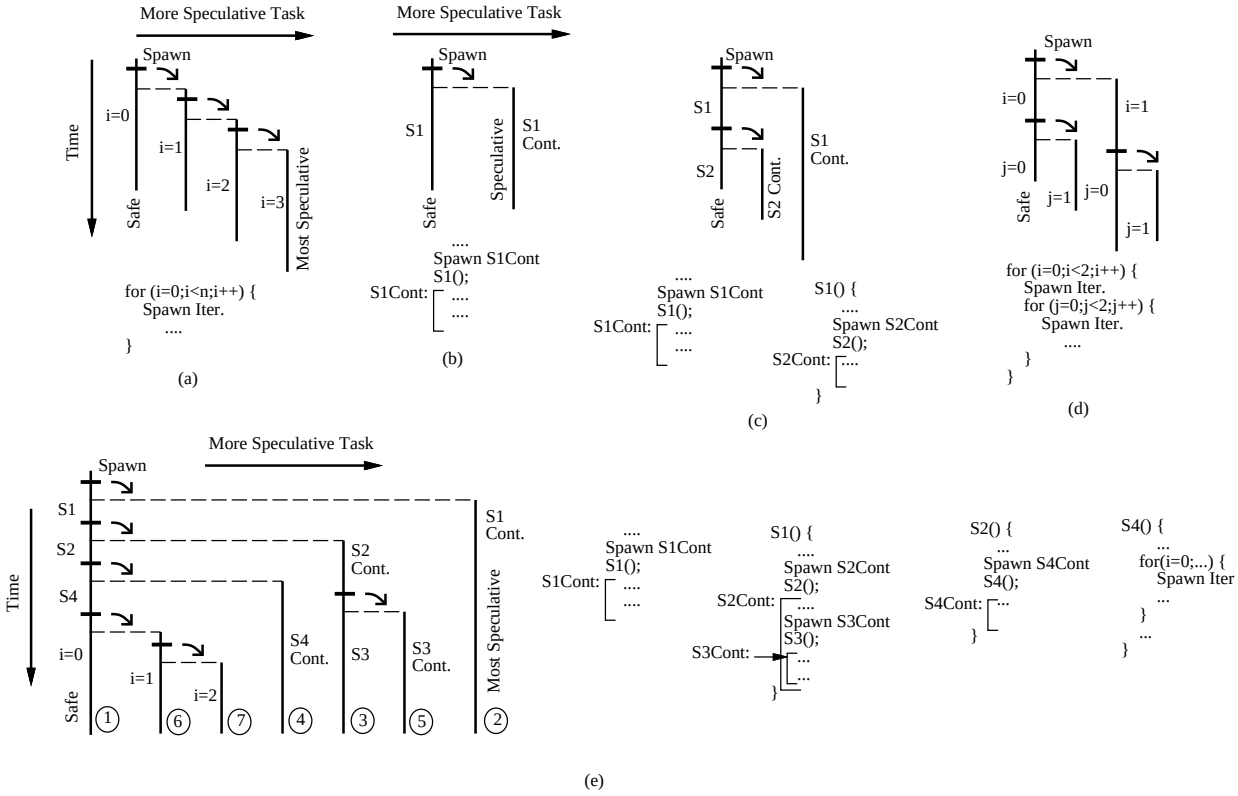


Figure 1: Task trees resulting from different approaches to build TLS tasks. In the figure, Cont and Iter mean continuation and iteration, respectively.

task spawns multiple tasks, the compiler inserts the spawns in strict reverse task order (last task is spawned first, etc). Second, the spawned tasks are less speculative than any task that was more speculative than their parent. These conventions are followed to make the spawn structure like that of nested loops and subroutines. Intuitively, these conventions are unlikely to limit the performance much, while they simplify the micro-architecture.

2.1 Why Supporting Out-of-Order Spawn in CMPs is Hard

Generally, with out-of-order spawn, *all* tasks can spawn, and parallelism expands in unexpected parts of the task tree at run time. As a result, in decentralized architectures such as CMPs, it becomes harder to maintain two cornerstones of TLS: task ordering and efficient resource allocation.

Task ordering is required in several TLS operations that are time-critical. Specifically, a task needs to know its immediate successor, to communicate the commit token or a squash signal. Moreover, any communication between two tasks requires knowing the tasks' relative order: such order determines whether a dependence violation is triggered, or what data version is returned to the requester. Unfortunately, when fine-grain tasks are spawned out of order, unpredictably and in different processors, high-speed ordering of tasks and its maintenance is hard.

Efficient allocation of resources (e.g. CPU or cache space) is crucial for TLS performance. Ideally, resources should be assigned to tasks that are safe or very likely to become so. However, with out-of-order spawning, there may be highly-speculative tasks that have been running for a long time. In this case, if the safe task wants to spawn and there are no free CPUs, should it kill the highly-speculative tasks? This is what past schemes do [8, 12]. Or should it abstain from spawning, do the work itself, and leave the highly-speculative tasks running?

Since decisions on task ordering and resource allocation have to be made very quickly, they need to be supported in the CMP micro-architecture. Given the complexity of TLS designs, however, such new micro-architecture needs to be simple.

2.2 Out-of-Order Spawning and Number of Processors

With out-of-order spawning, TLS can unlock additional parallelism: two code sections that are very separated in sequential execution can be executed *before* some of their intervening code sections have *even been spawned*. This feature enables more task overlap, and can benefit both machines with many processing elements (PE) and those with only few.

On the other hand, it is well-known that some integer applications have only modest coarse-grained parallelism. For example, for SpecInt, few-PE machines have often been a sweet spot. For these applications, even with out-of-order spawning, it is reasonable to target two-PE machines. Indeed, in code sections where, without out-of-order spawning, one of the two PEs of the machine would remain idle, we may now overlap the execution of two tasks that are far apart in sequential execution.

3 Novel Micro-Architectural Mechanisms

We propose three novel and simple micro-architectural mechanisms that, taken together, fundamentally enable high-speed tasking with out-of-order spawn in a TLS CMP. These mechanisms address the key issues of task ordering and efficient resource allocation, in an environment that is statically *unpredictable* (due to out-of-order spawn), *decentralized* (due to the CMP architecture), and has *no broadcast capabilities*. We enable high-speed task order management with *Splitting Timestamp Intervals* (Section 3.1) and the *Immediate Successor List* (Section 3.2). We enable high-speed decisions for efficient resource allocation with *Dynamic Task Merging* (Section 3.3). In the following, when we use the terms *successor* and *predecessor* task, we refer to sequential execution order.

3.1 Splitting Timestamp Intervals for Task Order Management

In any TLS system, tasks have a relative order, which they explicitly or implicitly embed in the CMP protocol messages they issue and the cached data they own. Such order is most obviously needed when two tasks communicate. For example, consider a task reading cached data produced by a second task. The relative order of the tasks is assessed, and the data is provided only if the former task is a successor of the latter. Similarly, consider an invalidation message from a task to data read by a second task. The task order is considered and, if the reader is a successor, a dependence violation is triggered.

Under in-order task spawn, recording task order is easy: since tasks are created in order, it suffices to assign monotonically increasing timestamps to newer tasks. A parent gives to its child its timestamp plus one. With this support, tasks with higher timestamps are successors of those with lower ones.

Unfortunately, such an approach does not work when tasks are created out of order. To maintain order now, we propose to represent a task with a *Timestamp Interval*, given by a *Base* and a *Limit* timestamp ({B,L}). Both base and limit timestamps are operated upon in a task spawn. Specifically, when a task spawns a child, it splits its timestamp interval in two pieces: the higher-range subinterval is given to the child (since it is more speculative), while the lower-range subinterval is kept by the parent. With this support, protocol messages and cached data are directly (or indirectly) associated with the base timestamp. When communication between tasks occurs, the base timestamps of the two tasks are compared *exactly* as in the in-order case.

As an example, Figure 2-(a) shows a program with a call to subroutine *S1*, which in turn calls *S2*. Assume that we use three tasks: task i executes the non-speculative code, j executes the continuation of *S1*, and k executes the continuation of *S2*. The resulting task tree is shown in Figure 2-(b), while Figure 2-(c) shows the timestamp intervals of each task.

The example assumes that the initial interval for task i is {B,L}, and that intervals are partitioned in half. When i spawns j, i keeps $\{B,\frac{L}{2}\}$ and j obtains $\{B+\frac{L}{2},\frac{L}{2}\}$. When i later spawns k, i retains $\{B,\frac{L}{4}\}$ and k obtains $\{B+\frac{L}{4},\frac{L}{4}\}$. With this scheme, as we move from safe to most speculative task following sequential order (i, k, and j), we encounter adjacent intervals ($\{B,\frac{L}{4}\}$, $\{B+\frac{L}{4},\frac{L}{4}\}$, $\{B+\frac{L}{2},\frac{L}{2}\}$) with increasing base timestamps.

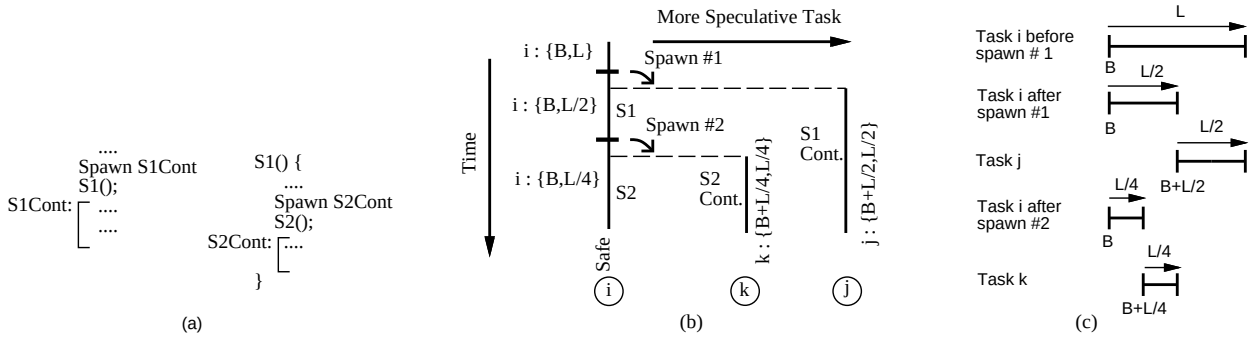


Figure 2: Changes in the base and limit timestamps when tasks are spawned. To save storage, the limit timestamp is encoded as an offset.

In general, a simple approach is to give $\frac{1}{2}$ of the current interval to the child. However, since a task rarely spawns more than a few other tasks, it makes sense to give a larger fraction of the interval to the child. In addition, there are two cases where we can be more efficient. The first one is when the parent knows that the child will not spawn any task; in this case, the parent can give it a single timestamp. The second case is when the parent knows that this is its last child; in this case, the parent can keep a single timestamp. These efficiencies may be obtainable with information gathered by the compiler or hardware predictors.

Our scheme assigns no L to the most speculative task, as it implicitly takes the maximum possible value (L_{max}). This allows the system to dynamically expand the range of used timestamps. Indeed, when the most speculative task spawns a child, it keeps the range {B,L_{max}} for itself, and sets the base of the child to B+L_{max}. The child is now the new most speculative task.

Finally, note that, in some cases, a task may reach a point where it needs to spawn a child and its interval has size 1. In addition, it is possible that a program exhausts the physically representable timestamp range. These infrequent cases are discussed in Section 4.2.

3.2 Immediate Successor List for Task Squash and Commit

In TLS, a task must be able to find its immediate successor very quickly, to perform the time-critical operations of commit and squash. Specifically, when the safe task commits, it passes the commit token to its immediate successor, which may be waiting for it to commit. As for squash, a task is squashed when it reads data prematurely (data violation) or is spawned in the wrong branch path (control violation). In the case of a control violation, the victim task receives a kill signal, which causes the destruction of any state modifications made by the task and terminates the task. In the case of a data violation, the victim task receives a restart signal, which induces the destruction of the state modifications and restarts task execution from its beginning – hoping that the re-execution will read correct data. In either case, a kill signal is also sent to the immediate successor of the victim task and, recursively, to the immediate successor of that one up until the most speculative task. This ensures that all possible side effects of the victim task are erased.

Under in-order task spawn, it is easy to find a task's immediate successor and, recursively,

immediate successors until the most speculative task. For example, consecutively spawned tasks are often allocated on contiguous processors, making it trivial to identify the immediate successor. In other designs, a table with immediate successor information is used, which is easy to maintain because only one task can spawn at a time. Finally, any scheme used is likely to be largely free of protocol races, as only one task spawns at a time.

Under out-of-order task spawn, identifying the immediate successor and all the more speculative tasks is not so straightforward. For example, in Figure 1-(e), if task *7* is killed, it is not trivial for it to identify and kill tasks *4*, *3*, *5*, and *2*, which were created before and independently of *7*. Moreover, any solution has to be carefully crafted to avoid inducing races in the TLS protocol of the distributed CMP if multiple operations happen concurrently.

To support efficient and race-free commit and squash, we propose that the tasks dynamically link themselves in hardware in a list according to their sequential order. We call this list the *Immediate Successor* (IS) list. To build the IS list, we add a hardware pointer to each task structure called the IS pointer. We leverage the fact that, at the time of the spawn, the child is always the immediate successor of its parent. Moreover, the child inherits the parent's immediate successor. Consequently, in our scheme, when a task spawns a child, the hardware gives the parent's IS to the child, and sets the parent's IS to point to the child. Moreover, when a task kills all its successors, the hardware sets its IS to nil. In the example of Figure 1-(e), the IS list links *1* to *6*, *6* to *7*, *7* to *4*, and so on. Task *2*'s IS pointer is nil.

With this support, when a task needs to pass the commit token, it uses the IS list. Moreover, when the victim task in a dependence violation needs to kill all its successors, it sends a kill signal with its own identity downstream the IS list. All successors are killed in turn. When the kill signal reaches a task with a nil IS, an acknowledgment is sent to the originating task, which sets its IS to nil. The result is very fast commit and squash. In addition, our proposal simplifies the TLS protocol implementation in a major way: even when multiple kill and commit signals occur concurrently, since all signals are serialized along the same path, the scheme minimizes protocol races.

3.3 Dynamic Task Merging for Efficient Resource Allocation

In TLS systems, tasks compete for CMP resources such as CPUs, on-chip contexts, and cache space. Under out-of-order task spawn, such competition is harder to manage than under in-order spawn. The reason is that highly-speculative tasks may hog resources and starve more critical (less speculative or even safe) tasks that are spawned later. For example, in Figure 1-(e), when safe task *1* is about to spawn *6*, all the CPUs and contexts in the CMP may be in use by more speculative tasks *4*, *3*, *5*, and *2*.

To allocate chip resources efficiently, we propose a new CMP microarchitectural technique that we call *Dynamic Task Merging*. It consists of transparent, hardware-driven merging of two consecutive tasks at run time. The merging may occur before or after the second task has been spawned. In effect, it enables the machine to prune some branches of the task tree based on

dynamic load conditions. The overall effects of dynamic task merging are an increase in the size of the running tasks and a reduction in their dynamic number.

These effects increase execution efficiency in several ways. First, highly-speculative tasks can be merged, therefore freeing resources for more critical tasks. Second, with large tasks, the overhead related to task spawn has a relatively lower weight, and both caches and branch predictors work better, as a CPU reuses their state for a longer time. Finally, given that the hardware can adjust the number of tasks at run time, the TLS compiler can be more aggressive at creating tasks, which may ultimately lead to higher performance.

Given a pair of tasks, we propose two types of dynamic task merging, depending on whether or not the second task has been spawned. If it has not, dynamic task merging typically involves skipping the spawn instruction of the second task and the task-end instruction of the first task. If the second task has already been spawned, dynamic task merging typically involves killing it and skipping the task-end instruction of the first task.

The first type of task merging can be triggered on any task when it is about to spawn a child. We call it *MergeNext*. The second type of task merging can be triggered on any pair of consecutive tasks in the CMP at any time. However, to maximize efficiency and simplify the implementation, we only trigger it on the two most speculative tasks in the CMP. Consequently, we call it *MergeLast*. Usually, we do it when a new task is about to be spawned somewhere in the CMP.

Note that *MergeNext* and *MergeLast* are not exclusive choices. Overall, every time that a task finds a spawn instruction, we select one of four possible choices: spawn normally, MergeNext, spawn and MergeLast, and both MergeNext and MergeLast. Figure 3 shows the choices when task *4* finds the spawn for *5*. In the rest of this section, we discuss the microarchitecture support for MergeNext and MergeLast, and the heuristics that we use to decide which of the four choices to select. Some compiler implementation details are discussed in Section 5.2.

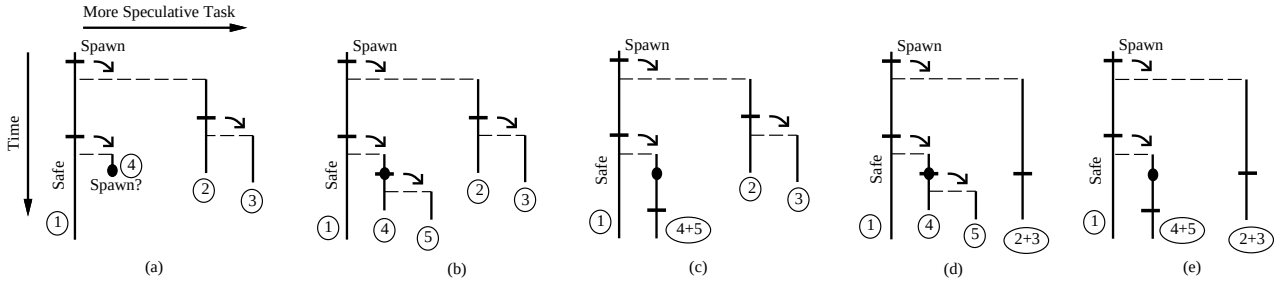


Figure 3: Choices when task *4* finds the spawn for *5*: spawn (b), MergeNext (c), spawn and MergeLast (d), and MergeNext and MergeLast (e).

3.3.1 MergeNext Microarchitecture

A task initiates a MergeNext by skipping a spawn instruction. After that, in the simplest case, the task will also have to skip the first task-end instruction that it finds, and finish only when it finds the second task-end. In general, if a task initiates N MergeNext operations by skipping N spawns,

it will have to also skip *N* task-ends and complete only when it finds the *N+1* one.

Consider now a task that skipped a spawn in a MergeNext and later spawns a child. In this case, since the child is more speculative, the responsibility to complete the task merge is "passed on" to the child: the child will skip the first task-end that it finds and finish only at the second one. As for the parent, it simply finishes at the first task-end that it finds.

The microarchitecture needed to support MergeNext is a counter in the processor called *Number of Ends to Skip* (NES). The NES belongs to the running task, and is checked and modified in hardware. Specifically, when a task initiates a MergeNext, the NES is incremented. When a task finds a task-end instruction, the NES is checked. If it is non-zero, it is decremented and the end instruction is skipped. Otherwise, the end is executed. Moreover, when a task spawns a child, its NES is copied to the child's and is then cleared. The child now owns the merges.

A task's NES is affected by two more events. First, when a task becomes the most speculative one (its IS pointer becomes nil), its NES ceases to matter — the task simply skips any task-end instruction that it finds. This is the appropriate behavior for the most speculative task, which should not be stopped by end instructions. However, if the task spawns a child, the NES of both tasks are updated as usual. The second special event occurs when a task gets restarted (Section 3.2). In this case as the task recovers its initial state, it also recovers its initial NES.

3.3.2 MergeLast Microarchitecture

MergeLast involves killing the most speculative task in the CMP and ensuring that, when the new most speculative task completes its own code, it executes the code of the killed task.

The microarchitecture needed to support MergeLast is the IS list (Section 3.2). A task initiates a MergeLast by sending a MergeLast hardware signal down the IS list. Each task in the list passes, in hardware, the signal and its own identity to its successor. When the signal reaches a task with a nil IS pointer, that task sends an acknowledgment to its immediate predecessor (whose identity it knows) and terminates. The immediate predecessor sets its IS pointer to nil, as it is now the most speculative task. No other action is necessary. When the latter task reaches its end, it will skip it and continue executing, effectively merging its code with that of the killed task. This is because, as discussed in Section 3.3.1, a task with a nil IS pointer skips task-ends.

Note that the operation of a task killing all its successors after a violation (Section 3.2) is similar to a MergeLast except that all the tasks downstream the IS list are killed. In fact, to keep the hardware simple, we implement such an operation as a set of MergeLast operations: the killing task keeps issuing MergeLast operations until it becomes the most speculative task.

3.4 Task Merge Heuristics

Every time that a task finds a spawn instruction, decisions on task merging are made. To keep the hardware simple and the overheads low, in this paper we propose a simple decision algorithm.

The algorithm is based on two notions. First, we conservatively assume that any running task,

even if highly speculative, is likely to perform useful work. Consequently, we try to avoid killing tasks. Second, we rely on squash information to reduce useless work. Specifically, if a task has been restarted twice due to violations, it is not allowed to get a CPU anymore. It simply remains in one of the several on-chip task contexts until it becomes safe. This policy prevents highly-speculative, frequently-squashed tasks from clogging the CPUs. It also allows the hardware to estimate the level of load in the CMP by examining the fraction of on-chip task contexts that are in use.

With this support, we use the following algorithm. We use the CPU usage to decide on MergeNext. If all CPUs are busy, since they appear to do useful work, we perform MergeNext. However, every *NumMNext* MergeNexts, we skip one to prevent tasks from becoming so large that a squash would be very costly.

As for MergeLast, we decide based on the estimated use of on-chip task contexts. If most of them are used, it is likely that many highly-speculative, frequently-squashed tasks are waiting. In this case, one could be killed with little performance penalty. While we could perform a MergeLast only when no context is free, the operation would then be in the critical path. Consequently, we use a threshold: if the estimated number of used contexts is over Th_{MLast} at the time of a spawn, we perform MergeLast.

4 Implementation Issues

To complete the architectural design for out-of-order tasking, this section discusses three related implementation details: task contexts, special cases in handling timestamp intervals, and scheduling tasks to CPUs.

4.1 Implementation of Task Contexts

Each processor has a table of task contexts, which keeps state for the tasks that are loaded on the processor. Of these tasks, only one is running at a time. Each context stores the following state for a task: {B,L} timestamp interval, IS pointer, NES, start PC of the task, and a pointer to a stack location with saved register state. This stack state is not read at the beginning of the task. Rather, it is read on a per-need basis. The context also has the Local ID (LID) associated to the task. As in many TLS systems, this LID is a short ID used to tag the cache lines accessed by the task. It acts as a form of indirection [16] that avoids the need to tag the lines with the whole B timestamp of the task.

The table of task containers is accessed by instructions such as spawn, and hardware signals such as restart or kill. Consider, for example, the case when a task must kill all its successors. In this case, the hardware passes the kill signal from the originating task down the IS list. For each task in the list, the operation is as follows. If the task is running, it is stopped. In all cases, the task's LID is marked as invalid, so that the task's cache lines become invalid and can be purged lazily. As in typical TLS systems, that LID remains unused until all its lines are purged from the cache; at that point, it can be reused.

If a task needs to be restarted, the initial PC and stack pointer are restored from the task

context. A new LID is then assigned.

4.2 Special Cases in Timestamp Intervals

There are two infrequent, special cases when handling timestamp intervals. The first one is when a task wants to spawn a child and has no interval to assign. In this case, it simply sends a kill signal down the IS list. This operation kills all successors, making the task the most speculative one. At this point, the task can obtain as many timestamps as needed (Section 3.1).

The second case is when a program exhausts the physically representable timestamp range. Our solution is to recycle old timestamps in chunks. For that, we divide the whole representable timestamp range into four chunks, based on the two most significant bits of B. When all the tasks with intervals in the lowest chunk (e.g. the 00 chunk) have committed, we recycle the chunk. This involves sending a reprogramming signal to the logic of the timestamp comparators so that timestamps in the recycled chunk are now the highest (i.e. 00 is more speculative than 11). Then, we can start assigning timestamps from the chunk to newer tasks.

The reprogramming signal is issued in the infrequent case that a task with an interval that straddles two chunks commits. With this approach, all the tasks in the CMP can at most use $\frac{3}{4}$ of the whole timestamp range at a time. To see how many tasks can be concurrently supported, assume that B and L have b and l bits, respectively. If, in the worst case, each task has a single child, and the child is given the maximum timestamp range possible (2^l), the maximum number of tasks is then $\frac{3}{4} \times 2^{b-l}$. Consequently, if we want to support about 20 concurrent tasks, $b-l$ should be at least 5.

4.3 Scheduling Tasks to CPUs

While all the tasks that have been spawned have their state loaded on on-chip task contexts, only as many tasks as CPUs can be running at a time. In practically all TLS proposals, tasks are scheduled strictly based on how speculative they are. Specifically, a less speculative task always preempts more speculative ones. Moreover, among the eligible tasks, the preempted one is the most speculative.

In practice, our evaluation will show that such a policy is an overkill, given the typical load and task sizes in our CMP, and our new task merging support. Consequently, we propose and use a simpler policy: we assign high priority to the non-speculative task, and a fixed low priority to all speculative tasks. There are no complex priorities and only the safe task can preempt.

4.4 Other Aspects

Most of the other aspects of a TLS CMP change little as we move from an in-order to an out-of-order spawning framework. For this reason and for brevity, we feel it is unnecessary to detail them. For example, our CMP uses a TLS protocol with lazy commit and multi-versioned L1 and L2 caches similar to [14]. As in that protocol, cache lines with speculative state cannot be displaced. If space is needed and the line is overwritten with a new address, the owner speculative task is sent a restart signal. When the task is re-scheduled again, it will restart. Context switches and

exceptions also cause restarts [14].

5 Compilation Support for Tasking with Out-of-Order Spawn

We have developed a full TLS compiler that generates in-order and out-of-order tasking out of sequential, integer applications. The compiler adds several passes to a still experimental branch of gcc 3.5. The branch uses a static single assignment tree as the high-level intermediate representation [6]. Building on this software allows us to leverage a complete compiler infrastructure. For example, we annotate the control flow graph structure with high-level information as we generate the tasks. Also, working at this high level is better than using a low-level representation such as RTL: we have better information and it is easier to perform pointer and dataflow analysis. At the same time, our transformations are much less likely to be affected by unwanted compiler optimizations than if we were working at the source-code level.

The resulting code quality, both when we enable and disable TLS, is comparable to the MIPSPro SGI compiler for integer codes at the O3 optimization level. This is because, in addition to using a much improved gcc version, we also use SGI's source-to-source optimizer (copt from MIPSPro). The latter performs PRE, loop unrolling, inlining, and other optimizations.

In the following, we highlight three issues: task generation, task merging, and profiling.

5.1 Task Generation and Hoisting

Our compiler extracts the following modules as individual tasks: subroutines from any nesting level, loop iterations from potentially multiple loops in a nest, and whole loops. All subroutines are extracted unless they are very small (in which case they are inlined) or they are libc functions that have system calls. Recursivity is handled seamlessly. In loop nests, the compiler makes decisions based on minimal loop iteration size.

As an example, Figure 4 shows how the compiler generates tasks out of a subroutine and its continuation. Chart (a) shows the dynamic execution into and out of the subroutine. The compiler first marks the subroutine and continuation as tasks, and inserts spawn statements (Chart (b)). After that, another compiler pass tries to hoist spawns. The goal is to get as much parallelism as possible, while making sure that the children of a task are spawned in reverse order, as discussed in Section 2. A spawn is hoisted as far up as we can, as long as the new position dominates the old one and both positions have execution equivalence [1]. We do not hoist past statements that can cause data or control dependence violations. Continuing with our example, Chart (c) hoists the continuation, while Chart (d) adds the hoisting of the subroutine. As usual, tasks on the right side are more speculative.

A final "task clean up" compiler pass looks for loops, subroutines, and iterations that were hoisted only a handful of instructions. In any such case, both the hoisting and the spawn instruction are eliminated, and the two corresponding tasks integrated into one. This pass eliminates

[1] We informally define execution equivalence as two blocks that are control equivalent and are executed the same number of times.

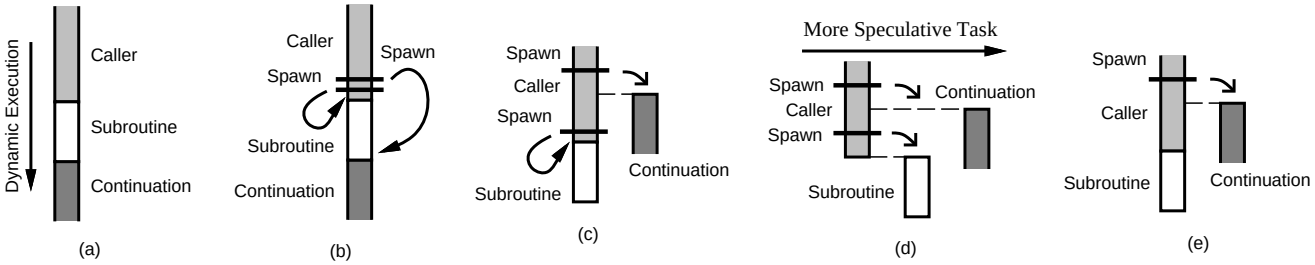


Figure 4: Generating tasks out of a subroutine and its continuation.

unnecessary spawn overheads. In the example, since the subroutine had little hoisting, the code is changed to Chart (e).

5.2 Task Merging

Task merging requires that, as a task completes its code, it goes on executing the code of its immediate successor. This means that the task must have a way of obtaining the live-in register values for its continuation code. In ordinary conditions, such code belongs to another task. However, with our compiler, a task can easily obtain the live-ins for its continuation and successor tasks. This is because all register values changed by a task that may be used by successors are sent to memory when the task finishes. Moreover, all the values needed by a task are read from memory.

5.3 Profiling Support

The compilation process for both in-order and out-of-order tasking includes running a simple profiler. The profiler takes the TLS executable and identifies those task spawn points that should be removed because they are likely to induce harmful squashes according to our models. The profiler returns the list of such spawns to the compiler. Then, the compiler generates the final TLS executable by removing these spawns and integrating their target tasks with the tasks' statically preceding code. On average, the profiler takes about two minutes to run.

The profiler executes the binaries sequentially, using the Train data set for SpecInt codes. As the profiler executes a task, it records the variables written. As it executes tasks that would be spawned earlier, it compares the addresses read against those written by predecessor tasks. With this, it can detect potential violations. The profiler also models a cache to estimate the number of misses in the real machine's L2, although no timing is modeled.

The profiler identifies those spawns where the ratio of squashes per task commit is higher than R_{squash}. For each of those spawns, it estimates the performance benefit that a task squash brings. Some benefit comes from the data prefetching provided by cache misses recorded before the task is squashed ($M_{squashed}$). Other benefit comes from true overlap of the instructions in the task with other tasks, as the task is re-executed after the violation ($I_{overlap}$). With these measurements, the profiler requests spawn removal if $T_I \times I_{overlap} + T_0 \times M_{squashed}$ is less than a threshold T_{perf}. In the formula, T_0 is the estimated average stall time per L2 miss, and T_I is the estimated execution time per instruction. The values for R_{squash}, T_0, T_I, and T_{perf} are listed in Section 6.

6 Evaluation Methodology

To evaluate TLS with out-of-order spawn, we use execution-driven simulations with detailed models of out-of-order superscalar processors and memory subsystems. The proposed architecture is a two-processor CMP with TLS support, which we call *TLS2*. Each processor in *TLS2* is a 4-issue core similar to a PowerPC 970. It has a private L1 cache that buffers multiversioned speculative data. Since an L1 cache may contain more than one version of the same line, we set its access time to a high value: 3 cycles. The L1 caches are connected through a crossbar to an on-chip shared L2 cache. The CMP uses a TLS coherence protocol with lazy task commit and multi-versioned L1 and L2 caches similar to [14]. One aspect of the protocol in [14] that is not supported is the overflow area.

For comparison purposes, we also model a chip with a single, very aggressive 8-issue processor. The chip has no TLS support. We call this architecture *8issue*. For this architecture, since the L1 does not support multiple versions, we set the L1 access time to a lower value: 2 cycles. Moreover, in our comparison, we use the same processor frequency for both *8issue* and *TLS2*. In a real implementation, the frequency of *TLS2* would be *higher* than *8issue*, therefore boosting the relative performance of *TLS2*. The complete set of parameters is shown in Table 1.

Processor Parameters	PROPOSED: *TLS2*	COMPETITION: *8issue*	Memory System and Tasking Parameters
Cores/chip	2	1	L2 cache size, assoc, line: 1 MB, 8, 64 B
Running tasks/core	1	1	L2 OC, RT: 2, 10
TLS hardware?	Yes	No	Mem bandwidth, RT: 8 GB/s, 400 cycles
Frequency	**5 GHz**	**5 GHz**	Task contexts/processor: 5
Fetch, issue, retire width	8, 4, 6	16, 8, 12	LIDs/processor: 128
ROB, I-window size	192, 96	360, 224	B, L timestamp size: 33, 28 bits
LD, ST queue	64, 48	128, 96	*NumMNext*: 10
Mem, int, fp units	2, 3, 2	4, 6, 4	Th_{MLast}: 16
Branch predictor:			Latency to kill mis-speculated task (min):
Penalty	17 cycles	17 cycles	From violation to proc
BTB	2 K, 2 way	2 K, 2 way	notification: 20 cycles
global gshare(11)	16 K	16 K	Time to drain proc pipeline: 17 cycles
local (2 bit)	16 K	16 K	Fraction of interval given to child: 3/4
L1 cache:			R_{squash}: 0.55
size, assoc, line	8 KB, 4, 32 B	8 KB, 4, 32 B	T_0: 390 cycles
OC, RT	1, **2**	1, **1**	T_I: 1 cycle
RT to neighbor's L1	7 cycles	—	T_{perf}: 100 cycles

Table 1: Architectures considered. In the table, OC and RT stand for occupancy and minimum-latency round trip from the processor, respectively. All cycle counts are in processor cycles. In our comparison, we use the same processor frequency for both *8issue* and *TLS2*.

In our experiments, we also use two more architectures built out of the 4-issue cores in *TLS2*: *4issue* and *TLS4*. *4issue* is a chip with a single core, one L1, one L2, and no TLS support. *TLS4* is an extension of *TLS2* that has 4 cores on chip; for simplicity, all parameters are the same as in *TLS2* except for the number of cores.

TLS2 uses as default the microarchitecture introduced in Sections 3 and 4.

Task spawn is performed by an instruction that takes the start PC of the child task. The instruction assembles a small packet that includes the child's start PC, timestamp interval, and

stack and IS pointers, as well as the parent's NES. In the meantime, to decide on task merging, the system estimates the load of the task contexts in the CMP, and if the other CPU is free. If it is decided to spawn on the other CPU, the packet is sent to it. Eventually, the child will execute there, accessing its live-ins through memory via the stack pointer.

Task commit is performed by an instruction that passes the commit token to its IS task. As for task context switch, most of the overhead is due to saving and restoring the registers. Note, however, that a task does not need to save all its 32 registers; only the ones marked as dirty are saved.

We drive our simulated architectures with the SpecInt 2000 applications running the Ref data set. We run all the SpecInt 2000 codes except four that either fail our compilation pass (*eon*, *gcc*, *perlbmk*), or cannot be run in our simulator (*vortex*). As shown in Table 2, we compare four different SpecInt binaries: unmodified binaries (*BaseApp*), TLS with in-order spawning (*InOrder*), and TLS with out-of-order spawning (*OutOrder*).

Name	TLS?	Description of Binary
BaseApp	N	Out-of-the-box, sequential version compiled with *O2*. No TLS instrumentation
OutOrder	Y	Our proposed out-of-order task spawning. Spawns to: (1) a procedure call (2) continuation of any procedure, and (3) iterations from multiple loops in nest
InOrder	Y	In-order task spawning. Selects the same tasks as *OutOrder*. Uses interprocedural analysis pass to eliminate tasks that violate the in-order spawning requirement.

Table 2: Versions of the SpecInt 2000 binaries executed.

These binaries are very different. Specifically, the TLS passes re-arrange the code into tasks and adds extra instructions for spawning and commit. In addition, these transformations obfuscate some conventional compiler optimizations, sometimes rendering them less effective. Consequently, to accurately compare the performance of the different binaries, we cannot simply time a fixed number of instructions. Instead, we insert "simulation markers" in the code, and simulate for a given number of markers. After skipping the initialization (typically 1-6 billion instructions), we execute up to a certain number of markers for all binaries, so that the *BaseApp* binary graduated more than 500 million instructions.

7 Evaluation

7.1 Execution Speedups

To evaluate our proposed support for TLS with out-of-order spawn, we compare the execution time of the in-order TLS binary *InOrder* and the *OutOrder* one running on the *TLS2* architecture. Of course, only the *OutOrder* binary can leverage our microarchitectural mechanisms. For comparison purposes, we also measure the execution times of the *BaseApp* binary running on the *4issue* and *8issue* architectures. The comparison to *4issue* shows the speedup of TLS relative to a single processor of the same size; the comparison to *8issue* shows the speedup of TLS relative to a much larger processor under the same frequency. Finally, we also evaluate *OutOrder* running on *TLS4*, to assess the effect of the number of processors in the CMP.

Figure 5 shows the speedups of the different binary-architecture combinations relative to

BaseApp running on *4issue*. For consistency, all the speedups in Section 7 are shown relative *BaseApp* running on *4issue*. The figure shows data for each application and the harmonic mean. On top of some of the bars, we show the average speedups. In addition, for the TLS bars, we show the ideal contribution of parallelism to the speedup as a black dot on each of the bars. The effect of parallelism on performance is discussed later.

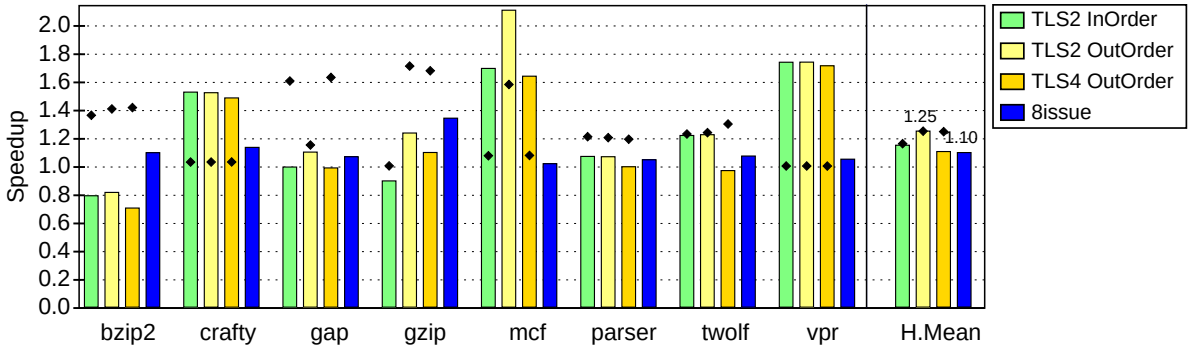


Figure 5: Speedups of different binary-architecture combinations relative to *BaseApp* running on *4issue*. The TLS bars also show the ideal contribution of parallelism to the speedup.

Consider first *OutOrder* on *TLS2* as compared to *InOrder* on *TLS2*. From the figure, we see that *TLS2 OutOrder* is equal-to or outperforms *BaseApp* on *4issue* for all of the applications. On average, it delivers a speedup of 1.25 over such an environment, while *InOrder* only attains an average speedup of 1.15. Two main factors give *OutOrder* the advantage. Firstly, *OutOrder* can select tasks from a larger number of locations in the program and can spawn the tasks earlier due to out of order hoisting. Better task selection enables *gzip* to attain a larger amount of parallelism, leading to good speedups. The second factor is our proposed microarchitectural mechanisms. Efficient hardware support for managing tasks lessens the penalty for restarting and killing tasks. Consequently, we are able to select tasks that provide good prefetching, as is the case with *mcf* which attains a superlinear speedup over executing on *4issue*. This is discussed more in Section 7.2. Overall, because the *OutOrder* binary can spawn tasks with fewer restrictions and benefits from the proposed microarchitectural mechanisms, it performs as well or better than *InOrder* on *TLS2*.

Consider now *OutOrder* running on *TLS2* or on *TLS4*. An increase in the number of processors on the CMP does not improve performance, rather, it decreases from 1.25 to 1.10, on average. When we distribute work across two more processors, we incur more overheads in the memory system and increase the likelihood of restarts because we spawn more tasks. Overall, *OutOrder* on *TLS4* is slightly worse, but is still an important design point for TLS systems because parallel and numerical applications can exploit larger numbers of processors.

Consider now *BaseApp* on *8issue*, which provides an average speedup of 1.10 over *4issue*. *TLS2* has an average speedup of 1.25 with *OutOrder*. Consequently, TLS with out-of-order spawning is able to attain a 1.14 speedup over a much larger processor operating at the same frequency. Overall, *OutOrder* on *TLS2* is competitive with a high frequency, wide-issue, microarchitecture.

7.2 Characterization

To understand the performance of *OutOrder* on *TLS2*, Table 3 provides some run-time measurements. Column 2 shows the average number of busy CPUs, which gives a sense for the degree of parallelism. On average, 1.4 CPUs are in use. These figures are small because of the limited parallelism present in SpecInt codes. However, not all the CPU activity is useful, since some tasks are squashed. Column 3 shows the fraction of busy cycles that execute non-squashed tasks. Fortunately, on average 93% of the work done by the CPUs is useful.

App	Busy CPUs	Useful/ Busy Cycles (%)	Extra Dynamic Instr (%)	Task Size (Instr)	Committed Out of Order Spawn (%)	# Spawns in Task Lifetime		# Events / Task Commit		
						No Merge	Merge	Merge Next	Merge Last	Restart
bzip2	1.708	82.40	12.435	22353	3.78	1.006	1.124	3.3488	0.0005	0.4064
crafty	1.032	99.84	9.451	3431	0.00	1.000	1.000	0.0002	0.0000	0.9980
gap	1.426	80.73	8.904	1401	0.00	1.352	1.001	0.7363	0.0000	0.9512
gzip	1.734	98.68	24.872	1884	0.07	1.271	1.000	3.8905	0.0000	0.2816
mcf	1.912	82.65	82.945	80	30.59	1.000	1.287	2.6479	0.2010	1.0234
parser	1.216	98.94	21.315	279	0.00	1.052	1.000	0.0699	0.0000	0.2382
twolf	1.239	99.97	29.493	115	0.00	1.000	1.000	0.0519	0.0000	0.0000
vpr	1.002	100.00	8.618	27311	0.000	1.000	1.001	0.2623	0.0000	0.0000
Avg	1.409	92.90	24.754	7107	4.30	1.085	1.052	1.3760	0.0252	0.4874

Table 3: Characterizing the run-time behavior of *OutOrder* on *TLS2*.

From the time spent by CPUs executing useful work, we can compute the *ideal* contribution of parallelism to TLS speedup. Figure 5 shows such a contribution for all TLS bars. Note that such a contribution is not equal to the distance between 1 and the top of the bars. The reason is that there are other effects that increase or decrease the speedup. Specifically, the speedup relative to *BaseApp* on *4issue* decreases due to at least two effects. First, the TLS environments have multiple L1 caches and processors on chip, which induces a higher number of cache misses and branch mispredictions, respectively. The second effect is the higher dynamic instruction count under TLS. The reason is the additional spawn, commit, and memory instructions, and the lower effectiveness of conventional compiler optimizations (Section 6). This effect can be seen in Column 4 of Table 3, which shows that TLS execution increases the dynamic instruction count by 25%.

On the other hand, TLS speedups can increase beyond that given by parallelism due to at least two factors. One is the data prefetching effect induced by speculative tasks, which bring into caches data that can later be used by other tasks. Prefetching can be beneficial to performance regardless of the degree of parallelism. The second factor is the additional resources on chip, which includes multiple BTBs and more functional units. However, the performance contribution due to additional resources will contribute only when parallelism can be exploited.

Overall, Figure 5 shows that some of these effects have at least as much importance as parallelism in TLS speedups on SpecInt applications. For example, in *mcf* using the *OutOrder* on *TLS2*, the parallelism is 1.58, while the speedup is 2.11. *mcf* obtains this speedup even in the presence of large overheads, as shown in Column 4 of Table 3. *mcf* also has a large percentage of wasted busy cycles which benefited execution via data prefetching. On the other hand, *gzip* exhibits a large degree of parallelism coupled with an average speedup. Since more than 98% of the work in *gzip* is useful, the speedup comes primarily from parallelism.

Column 5 of Table 3 shows the average number of graduated instructions in the tasks that

commit. On average, such tasks contain 7107 instructions. Given the frequent task merge operations, this is a small number of instructions. Obtaining large speedups with such small tasks is challenging.

The next few columns show parameters related to out-of-order spawning: the average number of tasks that are spawned out-of-order and successfully commit, and the average number of children spawned by a task in its lifetime (Columns 7-8) with and without the benefits of dynamic task merging. *mcf* uses the out-of-order hardware the best, with 30% of its committed tasks being spawned out-of-order. However, we can see from Column 6 that many benchmarks do not exploit out-of-order spawning when merging is enabled. These small percentages do not mean that out-of-order spawn should not be supported. Merging makes out-of-order spawning less frequent because we choose to skip a spawn instruction. Furthermore, out-of-order spawn is necessary to get speedups in *gap* and *gzip*. Without out-of-order spawn support, we could not select the tasks in *gzip* that provide good performance. *gzip* does benefit from out-of-order spawning even though the effect of merging reduces the number of spawns in a task lifetime from 1.3 down to 1.0. Overall, task merging decreases the number of spawns per task, thereby reducing the contribution of out-of-order spawns in the execution of the benchmark.

Finally, the last three columns show the frequency of key events in occurrences per task commit. We can see that all these events occur during execution. MergeNext is the most frequently occurring event of the three. For example, *gzip* performs a MergeNext approximately four times per task, providing it with one of the larger average task sizes. Its high percentage of useful work and large parallelism allows MergeNext more opportunities to occur per task. The MergeLast event occurs when resources need to be reclaimed and when a task needs to kill its successors. For most of the benchmarks, this event happens rarely, and for a number of benchmarks, it never occurred in our simulations. Finally, restart signals occur more frequently than MergeLast because restarts are necessary on any dependence violation. Even though a task is restarted half the time it is spawned, on average, it typically happens early in the task execution. This keeps the ratio of useful to busy cycles high.

7.3 Architecture Sensitivity Analysis

In this final section, we examine design variations. While we would like to assess the impact of each of our microarchitectural mechanisms separately, we cannot disable the IS list or the timestamp intervals because they are directly needed to support out-of-order spawn. However, we can measure the effect of disabling dynamic task merging. We also measure the impact of never running out of timestamps and of using an advanced task scheduling algorithm.

Figure 6 compares the speedup of our proposed *TLS2*, to *TLS2* with the individual changes mentioned. In all cases, the binary used is *OutOrder*. As usual, all the bars show speedups relative to *BaseApp* running on *4issue*. In the following, we consider each case in turn.

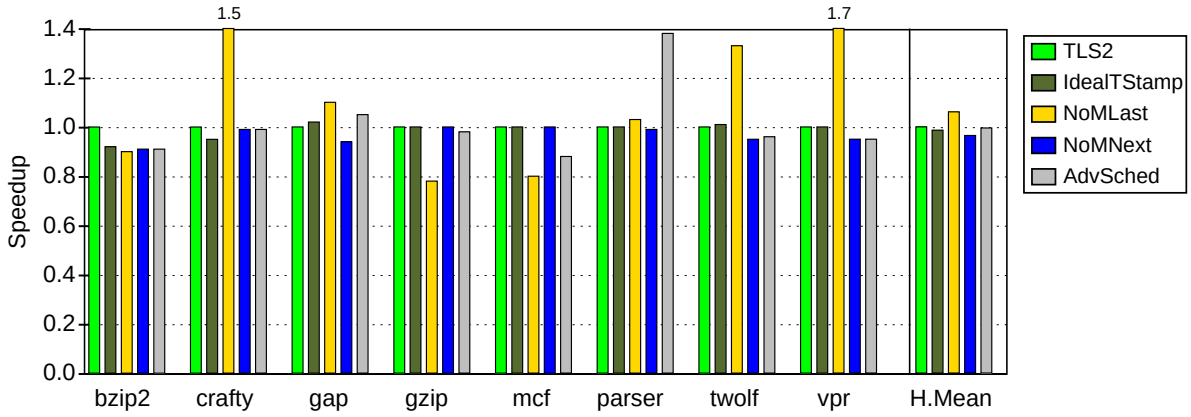


Figure 6: Comparing our proposed *TLS2* to *TLS2* with different architectural or software changes. In all cases, the binary used is *OutOrder*.

7.3.1 Disabling Dynamic Task Merging

The *NoMNext* and *NoMLast* bars in Figure 6 correspond to *TLS2* without MergeNext and without MergeLast, respectively. With *NoMNext*, the speedups are reduced by 4% on average. Disabling MergeNext benefits none of our simulated benchmarks. The main cause is the larger amount of work wasted due to squashes. Indeed, without MergeNext, tasks spawn more children. To assign on-chip contexts to all the less-speculative tasks, we have to squash more-speculative tasks frequently. As a result, *NoMNext* wastes more cycles on squashed tasks than *TLS2*. Consequently, we recommend supporting MergeNext.

With *NoMLast*, several programs, *gzip* for example, run much slower. The reason is that, typically, there are many highly-speculative tasks that use up all the task contexts, therefore preventing the non-speculative task from spawning. As a result, the CMP is using the resources poorly. A few benchmarks do gain from *NoMLast* because their average task size is increased and because the most speculative tasks are left running to either contribute to useful work or to provide prefetching for the less speculative threads. Overall, to unclog the CMP when necessary, we recommend supporting MergeLast.

7.3.2 Never Running out of Timestamps

In *TLS2*, when a task runs out of timestamps, it kills all its successors (Section 4.2). We would like to assess the overhead of these events. For that, bar *IdealTStamp* in Figure 6 corresponds to an ideal system with unlimited-sized timestamps and no additional overhead. Overall, the figure shows that *IdealTStamp* offers no performance advantage. The reason is that, in *TLS2*, running out of timestamps is relatively rare. In addition, *IdealTStamp* may perform worse than *TLS2* because restarts are occasionally beneficial to performance.

7.3.3 Supporting Advanced Task Scheduling

TLS2 uses a simple, two-priority algorithm to schedule tasks to CPUs (Section 4.3). In bar *AdvSched* of Figure 6, we change it to support the advanced scheduling outlined in Section 4.3: tasks are

strictly prioritized and preempt each other based on how speculative they are. Moreover, we set the scheduling overhead to zero.

We see that *AdvSched* delivers no advantage. The main reason is that, in *TLS2*, MergeNext regulates the number of ready tasks effectively. As a result, starvation of tasks by more speculative ones occurs with tolerable frequency.

8 Related Work

There are three pieces of work on environments that need out-of-order spawning.

Hammond *et al.* [8] have proposed a TLS CMP that supports out-of-order spawn with both subroutine and loop-iteration tasks. Their scheme is very different than ours. Each processor has a co-processor that controls TLS mechanisms by running software handlers. There are 2 broadcast busses. Co-processors are informed of what task is running on what processor. Messages are snooped from the broadcast busses and, based on their source, the co-processors can tell the relative ordering. Since caches contain state from a single task, no task ID is necessary. Squash signals are also broadcast. Commits require access to a centralized software data structure in shared memory. The most speculative task is killed if there is no space in the CMP. Overall, this is a broadcast-based, relatively centralized architecture. Moreover, the authors conclude that their scheme has too much control software overhead to support subroutine tasks. Their finding motivates our work.

There are several high-level performance-evaluation studies of environments that need out-of-order spawning [12, 13, 21, 22]. They often assume some ideal architectural feature, such as an infinite number of processors or perfect value prediction, and compare the performance to more realistic environments. Of those, [12, 13] examine a variety of sources of parallelism, including iterations from multiple loop levels, any-nesting subroutines, and full loops. [21, 22] examine subroutine-level nested parallelism in detail. None of these papers has attempted to describe the design of micro-architectural structures to support the tasks used. Consequently, they have not addressed the problems we cover. Our paper is the first detailed microarchitectural design of high-speed out-of-order tasking on a CMP and its evaluation. Many of the problems we solve do not even appear in these previous studies (e.g. task ID limitations or fast access to immediate successor).

DMT is a centralized, SMT-like processor whose hardware can extract out-of-order tasks from unmodified binaries [1]. The design is very different than ours because, being an SMT, it uses centralized structures that are *unusable* in a CMP. Specifically, DMT has a centralized hardware tree that records which tasks are successors of which. To determine the order of two tasks, the hardware walks the tree when: (1) there is a collision in the centralized LD/ST queue, or (ii) a task commits and needs to verify the register predictions for successor tasks. This centralization means that DMT does not need our proposed IS list and timestamp intervals. DMT kills the most speculative task if there is no space in the processor. However, it requires no analysis of matching task-end and spawn instruction like us because the binary is unmodified. Moreover, DMT does not support MergeNext, which is our new way of dynamically managing the resources in the system.

9 Conclusion

For CMPs with TLS to deliver on their promise, they must support dynamic environments where tasks are spawned out-of-order and unpredictably. Given the decentralized hardware of a CMP, this is challenging. This paper has been the first one to identify and design micro-architectural mechanisms that, taken together, fundamentally enable high-speed tasking with out-of-order spawn in a TLS CMP. We proposed three simple primitives for correct and fast task ordering and resource allocation. Task ordering is enabled with Splitting Timestamp Intervals for low-overhead order management, and with the Immediate Successor List for efficient task squash and commit. Fast and efficient resource allocation is enabled with Dynamic Task Merging, which directs speculative parallelism to the most beneficial code sections. To evaluate these primitives, we developed a TLS compiler with out-of-order spawn. With our mechanisms, a CMP with 2 4-issue processors increases the average speedup of SpecInt 2000 applications from 1.15 (no out-of-order spawn) to 1.25 (out-of-order spawn). Moreover, the resulting CMP outperforms an 8-issue superscalar: with the same clock frequency, the CMP delivers an average speedup of 1.14 over the 8-issue processor.

Overall, with our micro-architectural mechanisms, a CMP can leverage more sources of parallelism and boost TLS speedups. Note that, for applications that can exploit more PEs in the CMP (such as numerical applications), our technology is more enabling. The reason is that the codes can use our new hardware better. Finally, we expect that, as we improve our compiler algorithms, TLS speedups will improve.

References

[1] H. Akkary and M. Driscoll. A Dynamic Multithreading Processor. In *International Symposium on Microarchitecture*, pages 226–236, December 1998.

[2] A. Bhowmik and M. Franklin. A General Compiler Framework for Speculative Multithreading. In *Proceedings of 14th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, August 2002.

[3] M Chen and K. Olukotun. Exploiting Method-Level Parallelism in Single-Threaded Java Programs. In *Proceedings of the 7th International Conference on Parallel Architectures and Compilation Techniques (PACT'98)*, October 1998.

[4] M Chen and K. Olukotun. The Jrpm System for Dynamically Parallelizing Java Programs. In *Proceedings of the 30th International Symposium on Computer Architecture*, June 2003.

[5] P. S. Chen, M. Y. Hung, Y. S. Hwang, R. D. Ju, and J. K. Lee. Compiler Support for Speculative Multithreading Architecture with Probabilistic Points-to Analysis. In *Proceedings of the 2003 Symposium on Principles and Practice of Parallel Programming (PPoPP'03)*, pages 25–36, June 2003.

[6] SSA for trees - GNU project. URL, May 2003. "http://www.gccsummit.org/2003/view_abstract.php?talk=2".

[7] S. Gopal, T. Vijaykumar, J. Smith, and G. Sohi. Speculative Versioning Cache. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, pages 195–205, February 1998.

[8] L. Hammond, M. Willey, and K. Olukotun. Data Speculation Support for a Chip Multiprocessor. In *8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 58–69, October 1998.

[9] V. Krishnan and J. Torrellas. A Chip-Multiprocessor Architecture with Speculative Multithreading. *IEEE Trans. on Computers*, pages 866–880, September 1999.

[10] X. F. Li, Z. H. Dui, Q. Y. Zhao, and T. F. Ngai. Software Value Prediction for Speculative Parallel Threaded Computations. In *First Value Prediction Workshop*, pages 18–25, June 2003.

[11] P. Marcuello and A. Gonzalez. Clustered Speculative Multithreaded Processors. In *Proceedings of the 1999 International Conference on Supercomputing*, pages 365–372, June 1999.

[12] P. Marcuello and A. Gonzalez. A Quantitative Assessment of Thread-level Speculation Techniques. In *Proceedings of the 14th International Parallel and Distributed Processing Symposium (IPDPS'00)*, pages 595–604, 2000.

[13] Jeffrey T. Oplinger, David L. Heine, and Monica S. Lam. In Search of Speculative Thread-Level Parallelism. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT'99)*, October 1999.

[14] M. Prvulovic, M. J. Garzarán, L. Rauchwerger, and J. Torrellas. Removing architectural bottlenecks to the scalability of speculative parallelization. In *Proceedings of the 28th International Symposium on Computer Architecture (ISCA'01)*, pages 204–215, June 2001.

[15] G.S. Sohi, S.E. Breach, and T.N. Vijayakumar. Multiscalar Processors. In *22nd International Symposium on Computer Architecture*, pages 414–425, June 1995.

[16] J. Steffan, C. Colohan, A. Zhai, and T. Mowry. A Scalable Approach to Thread-Level Speculation. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 1–12, June 2000.

[17] M. Tremblay. MAJC: Microprocessor Architecture for Java Computing. Hot Chips, August 1999.

[18] J. Tsai, J. Huang, C. Amlo, D. Lilja, and P. Yew. The Superthreaded Processor Architecture. *IEEE Trans. on Computers*, 48(9):881–902, September 1999.

[19] J. Y. Tsai, Z. Jiang, and P. C. Yew. Compiler Techniques for the Superthreaded Architecture. In *International Journal of Parallel Programming*, pages 27(1):1–19, 1999.

[20] T. Vijaykumar and G. Sohi. Task Selection for a Multiscalar Processor. In *Proceedings of the 31th Annual International Symposium on Microarchitecture*, pages 81–92, November 1998.

[21] F. Warg and P.Stenstrom. Improving Speculative Thread-Level Parallelism Through Module Run-Length Prediction. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS'03)*, 2003.

[22] F. Warg and P. Stenström. Limits on Speculative Module-Level Parallelism in Imperative and Object-Oriented Programs on CMP Platforms. In *Proceedings of the 10th International Conference on Parallel Architectures and Compilation Techniques (PACT'01)*, September 2001.

[23] A. Zhai, C. Colohan, J. Steffan, and T. Mowry. Compiler Optimization of Scalar Value Communication Between Speculative Threads. In *ASPLOS X Proceedings*, San Jose, CA, October 2002.

[24] C. Zilles and G. Sohi. Master/Slave Speculative Parallelization. In *Proceedings of the 35th Annual International Symposium on Microarchitecture*, pages 65–77, November 2002.

Cherry: Checkpointed Early Resource Recycling in Out-of-order Microprocessors*

José F. Martínez Jose Renau† Michael C. Huang‡ Milos Prvulovic† Josep Torrellas†

Computer Systems Laboratory, Cornell University
martinez@csl.cornell.edu

†Dept. of Computer Science, University of Illinois at Urbana-Champaign
{renau,prvulovi,torrellas}@cs.uiuc.edu

‡Dept. of Electrical and Computer Engineering, University of Rochester
michael.huang@ece.rochester.edu

ABSTRACT

This paper presents *CHeckpointed Early Resource RecYcling* (*Cherry*), a hybrid mode of execution based on ROB and checkpointing that decouples resource recycling and instruction retirement. Resources are recycled early, resulting in a more efficient utilization. Cherry relies on state checkpointing and rollback to service exceptions for instructions whose resources have been recycled. Cherry leverages the ROB to (1) not require in-order execution as a fallback mechanism, (2) allow memory replay traps and branch mispredictions without rolling back to the Cherry checkpoint, and (3) quickly fall back to conventional out-of-order execution without rolling back to the checkpoint or flushing the pipeline.

We present a Cherry implementation with early recycling at three different points of the execution engine: the load queue, the store queue, and the register file. We report average speedups of 1.06 and 1.26 in SPECint and SPECfp applications, respectively, relative to an aggressive conventional architecture. We also describe how Cherry and speculative multithreading can be combined and complement each other.

1 INTRODUCTION

Modern out-of-order processors typically employ a reorder buffer (ROB) to retire instructions in order [18]. In-order retirement enables precise bookkeeping of the architectural state, while making out-of-order execution transparent to the user. When, for example, an instruction raises an exception, the ROB continues to retire instructions up to the excepting one. At that point, the processor's architectural state reflects all the updates made by preceding instructions, and none of the updates made by the excepting instruction or its successors. Then, the exception handler is invoked.

One disadvantage of typical ROB implementations is that individual instructions hold most of the resources that they use until they retire. Examples of such resources are load/store queue entries and physical registers [1, 6, 21, 23]. As a result, an instruction that completes early holds on to these resources for a long time, even if it does not need them anymore. Tying up unneeded resources limits performance, as new instructions may find nothing left to allocate.

To tackle this problem, we propose *CHeckpointed Early Resource RecYcling* (*Cherry*). Cherry is a mode of execution that decouples the recycling of the resources used by an instruction and the retirement of the instruction. Resources are released early and gradually and, as a result, they are utilized more efficiently. For a processor with a given level of resources, Cherry's early recycling can boost the performance; alternatively, Cherry can deliver a given level of performance with fewer resources.

While Cherry uses the ROB, it also relies on state checkpointing to roll back to a correct architectural state when exceptions arise for instructions whose resources have already been recycled. When this happens, the processor re-executes from the checkpoint in conventional *out-of-order* mode (non-Cherry mode). At the time the exception re-occurs, the processor handles it precisely. Thus, Cherry supports precise exceptions. Moreover, Cherry uses the cache hierarchy to buffer memory system updates that may have to be undone in case of a rollback; this allows much longer checkpoint intervals than a mechanism limited to a write buffer.

At the same time, Cherry leverages the ROB to (1) not require in-order execution as a fallback mechanism, (2) allow memory replay traps and branch mispredictions without rolling back to the Cherry checkpoint, and (3) quickly fall back to conventional out-of-order execution without rolling back to the checkpoint or even flushing the pipeline.

To illustrate the potential of Cherry, we present an implementation on a processor with separate structures for the instruction window, ROB, and register file. We perform early recycling at three key points of the execution engine: the load queue, the store queue, and the register file. To our knowledge, this is the first proposal for early recycling of load/store queue entries in processors with load speculation and replay traps. Overall, this Cherry implementation results in average speedups of 1.06 for SPECint and 1.26 for SPECfp applications, relative to an aggressive conventional architecture with an equal amount of such resources.

Finally, we discuss how to combine Cherry and Speculative Multithreading (SM) [4, 9, 14, 19, 20]. These two checkpoint-based techniques complement each other: while Cherry uses potentially unsafe resource recycling to enhance instruction overlap *within* a thread, SM uses potentially unsafe parallel execution to enhance instruction overlap *across* threads. We demonstrate how a combined scheme reuses much of the hardware required by either technique.

This paper is organized as follows: Section 2 describes Cherry in detail; Section 3 explains the three recycling mechanisms used in this work; Section 4 presents our setup to evaluate Cherry; Section 5 shows the evaluation results; Section 6 presents the integration of Cherry and SM; and Section 7 discusses related work.

*This work was supported in part by the National Science Foundation under grants CCR-9970488, EIA-0081307, EIA-0072102, and CHE-0121357; by DARPA under grant F30602-01-C-0078; and by gifts from IBM, Intel, and Hewlett-Packard.

2 CHERRY: CHECKPOINTED EARLY RESOURCE RECYCLING

The idea behind Cherry is to decouple the recycling of the resources consumed by an instruction and the retirement of the instruction. A Cherry-enabled processor recycles resources as soon as they become unnecessary in the normal course of operation. As a result, resources are utilized more efficiently. Early resource recycling, however, can make it hard for a processor to achieve a consistent architectural state if needed. Consequently, before a processor enters Cherry mode, it makes a checkpoint of its architectural registers in hardware (Section 2.1). This checkpoint may be used to roll back to a consistent state if necessary.

There are a number of events whose handling requires gathering a precise image of the architectural state. For the most part, these events are memory replay traps, branch mispredictions, exceptions, and interrupts. We can divide these events into two groups:

The first group consists of memory replay traps and branch mispredictions. A memory replay trap occurs when a load is found to have issued to memory out of order with respect to an older memory operation that overlaps [1]. When the event is identified, the offending load and all younger instructions are re-executed (Section 3.1.1). A branch misprediction squashes all instructions younger than the branch instruction, after which the processor initiates the fetching of new instructions from the correct path.

The second group of events comprises exceptions and interrupts. In this paper we use the term *exception* to refer to any *synchronous* event that requires the precise architectural state at a particular instruction, such as a division by zero or a page fault. In contrast, we use *interrupt* to mean *asynchronous* events, such as I/O or timer interrupts, which are not directly associated with any particular instruction.

The key aspect that differentiates these two groups is that, while memory replay traps and branch mispredictions are a common, direct consequence of ordinary speculative execution in an aggressive out-of-order processor, interrupts and exceptions are extraordinary events that occur relatively infrequently.

As a result, the philosophy of Cherry is to allow early recycling of resources *only* when they are not needed to support (the relatively common) memory replay traps and branch mispredictions. However, recycled resources may be needed to service extraordinary events, in which case the processor restores the checkpointed state and restarts execution from there (Section 2.3.3).

To restrict resource recycling in this way, we identify a ROB entry as the *Point of No Return* (*PNR*). The PNR corresponds to the oldest instruction that can still suffer a memory replay trap or a branch misprediction (Figure 1). Early resource recycling is allowed only for instructions older than the PNR.

Instructions that are no older than the PNR are called *reversible*. In these instructions, when memory replay traps, branch mispredictions, or exceptions occur, they are handled as in a conventional out-of-order processor. It is never necessary to roll back to the checkpointed state. In particular, exceptions raised by reversible instructions are precise.

Instructions that are older than the PNR are called *irreversible*. Such instructions may or may not have completed their execution. However, some of them may have released their resources. In the event that an irreversible instruction raises an exception, the processor has to roll back to the checkpointed state. Then, the processor executes in conventional *out-of-order* mode (non-Cherry or normal mode) until the exception re-occurs. When the exception re-occurs, it is handled in a *precise* manner as in a conventional processor. Then, the processor can return to Cherry mode if desired (Section 2.3.3).

As for interrupts, because of their asynchronous nature, they are always handled without any rollback. Specifically, processor execution seamlessly falls back to non-Cherry mode (Section 2.1). Then, the interrupt is processed. After that, the processor can return to Cherry mode.

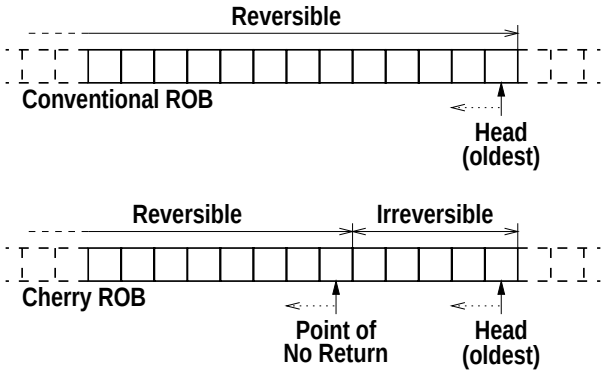


Figure 1: Conventional ROB and Cherry ROB with the Point of No Return (PNR). We assume a circular ROB implementation with Head and Tail pointers [18].

The position of the PNR depends on the particular implementation and the types of resources recycled. Conservatively, the PNR can be set to the oldest of (1) the oldest unresolved branch instruction (U_B), and (2) the oldest memory instruction whose address is still unresolved (U_M). Instructions older than oldest(U_B, U_M) are not subject to reply traps or squashing due to branch misprediction. If we define U_L and U_S as the oldest load and store instruction, respectively, whose address is unresolved, the PNR expression becomes oldest(U_B, U_L, U_S). In practice, a more aggressive definition is possible in some implementations (Section 3).

In the rest of this section, we describe Cherry as follows: first, we introduce the basic operation under Cherry mode; then, we describe needed cache hierarchy support; next, we address events that cause the squash and re-execution of instructions; finally, we examine an important Cherry parameter.

2.1 Basic Operation under Cherry Mode

Before a processor can enter Cherry mode, a checkpoint of the architectural register state has to be made. A simple support for checkpointing includes a backup register file to keep the checkpointed register state and a retirement map at the head of the ROB. Of course, other designs are possible, including some without a retirement map [23].

With this support, creating a checkpoint involves copying the architectural registers pointed to by the retirement map to the backup register file, either eagerly or lazily. If it is done eagerly, the copying can be done in a series of bursts. For example, if the hardware supports four data transfers per cycle, 32 architectural values can be backed up in eight processor cycles. Note that the backup registers are not designed to be accessed by conventional operations and, therefore, they are simpler and take less silicon than the main physical registers. If the copying is done lazily, the physical registers pointed to by the retirement map are simply tagged. Later, each of them is backed up before it is overwritten.

While the processor is in Cherry mode, the PNR races ahead of the ROB head (Figure 1), and early recycling takes place in the irreversible set of instructions. As in non-Cherry mode, the retirement map is updated as usual as instructions retire. Note, however, that the retirement map may point to registers that have already been recycled and used by other instructions. Consequently, the true architectural state is unavailable—but reconstructible, as we explain

below.

Under Cherry mode, the processor boosts the IPC through more efficient resource utilization. However, the processor is subject to exceptions that may cause a costly rollback to the checkpoint. Consequently, we do not keep the processor in Cherry mode indefinitely. Instead, at some point, the processor falls back to non-Cherry mode.

This can be accomplished by simply freezing the PNR. Once all instructions in the irreversible set have retired, and thus the ROB head has caught up with the PNR, the retirement map reflects the true architectural state. By this time, all the resources that were recycled early would have been recycled in non-Cherry mode too. This *collapse step* allows the processor to fall back to non-Cherry mode smoothly. Overall, the checkpoint creation, early recycling, and collapse step is called a *Cherry cycle*.

Cherry can be used in two ways. One way is to enter Cherry mode only as needed, for example when the utilization of one of the resources (physical register file, load queue, etc.) reaches a certain threshold. This situation may be caused by an event such as a long-latency cache miss. Once the pressure on the resources falls below a second threshold, the processor returns to non-Cherry mode. We call this use *on-demand Cherry*.

Another way is to run in Cherry mode continuously. In this case, the processor kepts executing in Cherry mode irrespective of the regime of execution, and early recycling takes place all the time. However, from time to time, the processor needs to take a new checkpoint in order to limit the penalty of an exception in an irreversible instruction. To generate a new checkpoint, we simply freeze the PNR as explained before. Once the collapse step is completed, a new checkpoint is made, and a new Cherry cycle starts. We call this use *rolling Cherry*.

2.2 Cache Hierarchy Support

While in Cherry mode, the memory system receives updates that have to be discarded if the processor state is rolled back to the checkpoint. To support long Cherry cycles, we must allow such updates to overflow beyond the processor buffers. To make this possible, we keep all these processor updates within the local cache hierarchy, disallowing the spill of any such updates into main memory. Furthermore, we add one *Volatile* bit in each line of the local cache hierarchy to mark the updated lines.

Writes in Cherry mode set the Volatile bit of the cache line that they update. Reads, however, are handled as in non-Cherry mode.[1] Cache lines with the Volatile bit set may not be displaced beyond the outermost level of the local cache hierarchy, e.g. L2 in a two-level structure. Furthermore, upon a write to a cached line that is marked dirty but not Volatile, the original contents of the line are written back to the next level of the memory hierarchy, to enable recovery in case of a rollback. The cache line is then updated, and remains in state dirty (and now Volatile) in the cache.

If the processor needs to roll back to the checkpoint while in Cherry mode, all cache lines marked Volatile in its local cache hierarchy are gang-invalidated as part of the rollback mechanism. Moreover, all Volatile bits are gang-cleared. On the other hand, if the processor successfully falls back to non-Cherry mode, or if it creates a new checkpoint while in rolling Cherry, we simply gang-clear the Volatile bits in the local cache hierarchy.

These gang-clear and gang-invalidation operations can be done in a handful of cycles using inexpensive custom circuitry. Figure 2 shows a bit cell that implements one Volatile bit. It consists of a standard 6-T SRAM cell with one additional transistor for gang-clear (inside the dashed circle). Assuming a 0.18μm TSMC process,

[1] Read misses that find the requested line marked Volatile in a lower level of the cache hierarchy also set (inherit) the Volatile bit. This is done to ensure that the lines with updates are correctly identified in a rollback.

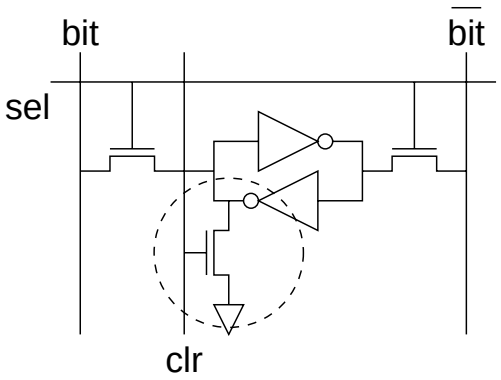


Figure 2: Example of the implementation of a Volatile bit with a typical 6-T SRAM cell and an additional transistor for gang-clear (inside the dashed circle).

and using SPICE to extract the capacitance of a line that would gang-clear 8Kbits (one bit per 64B cache line in a 512KB cache), we estimate that the gang-clear operation takes 6-10 FO4s [13]. If the delay of a processor pipeline stage is about 6-8 FO4s, gang-clearing can be performed in about two processor cycles. Gang-invalidation simply invalidates all lines whose Volatile bit is set (by gang-clearing their Valid bits).

Finally, we consider the case of a cache miss in Cherry mode that cannot be serviced due to lack of evictable cache lines (all lines in the set are marked Volatile). In general, if no space can be allocated, the application must roll back to the checkpoint. To prevent this from hapenning too often, one may bound the length of a Cherry cycle, after which a new checkpoint is created. However, a more flexible solution is to include a fully associative victim cache in the local cache hierarchy, that accommodates evicted lines marked Volatile. When the number of Volatile lines in the victim cache exceeds a certain threshold, an interrupt-like signal is sent to the processor. As with true interrupts (Section 2.3.2), the processor proceeds with a collapse step and, once in non-Cherry mode, gang-clears all Volatile bits. Then, a new Cherry cycle may begin.

2.3 Squash and Re-Execution of Instructions

The main events that cause the squash and possible re-execution of in-flight instructions are memory replay traps, branch mispredictions, interrupts, and exceptions. We consider how each one is handled in Cherry mode.

2.3.1 Memory Replay Traps and Branch Mispredictions

Only instructions in the reversible set (Figure 1) may be squashed due to memory replay traps or branch mispredictions. Since resources have not yet been recycled in the reversible set, these events can be handled in Cherry mode conventionally. Specifically, in a replay trap, the offending load and all the subsequent instructions are replayed; in a branch misprediction, the instructions in the wrong path are squashed and the correct path is fetched.

2.3.2 Interrupts

Upon receiving an interrupt while in Cherry mode, the hardware automatically initiates the transition to non-Cherry mode by entering a collapse step. Once non-Cherry mode is reached, the processor handles the interrupt as usual.

Note that Cherry handles interrupts *without rolling back to the checkpoint*. The only difference with respect to a conventional processor is that the interrupt may have a slightly higher response time. Depending on the application, we estimate the increase in the response time to range from tens to a few hundred nanoseconds. Such an increase is tolerable for typical asynchronous interrupts. In the unlikely scenario that this extra latency is not acceptable, the simplest solution is to disable Cherry.

2.3.3 Exceptions

A processor running in Cherry mode handles all exceptions precisely. An exception is processed differently depending on whether it occurs on a reversible or an irreversible instruction (Figure 1). When it occurs on a reversible one, the corresponding ROB entry is marked. If the instruction is squashed before the PNR gets to it (e.g. it is in the wrong path of a branch), the (false) exception will have no bearing on Cherry. If, instead, the PNR reaches that ROB entry while it is still marked, the processor proceeds to exit Cherry mode (Section 2.1): the PNR is frozen and, as execution proceeds, the ROB head eventually catches up with the PNR. At that point, the processor is back to non-Cherry mode and, since the excepting instruction is at the ROB head, the appropriate handler is invoked.

If the exception occurs on an irreversible instruction, the hardware automatically rolls back to the checkpointed state and restarts execution from there in non-Cherry mode. Rolling back to the checkpointed state involves aborting any outstanding memory operations, gang-invalidating all cache lines marked Volatile, gang-clearing all Volatile bits, restoring the backup register file, and starting to fetch instructions from the checkpoint. The processor executes in conventional out-of-order mode (non-Cherry mode) until the exception re-occurs. At that point, the exception is processed normally, after which the processor can re-enter Cherry mode.

It is possible that the exception does not re-occur. This may be the case, for example, for page faults in a shared-memory multiprocessor environment. Consequently, we limit the number of instructions that the processor executes in non-Cherry mode before returning to Cherry mode. One could remember the instruction that caused the exception and only re-execute in non-Cherry mode until such instruction retires. However, a simpler, conservative solution that we use is to remain in non-Cherry mode until we retire the number of instructions that are executed in a Cherry cycle. Section 2.4 discusses the optimal size of a Cherry cycle.

2.3.4 OS and Multiprogramming Issues

Given that the operating system performs I/O mapped updates and other hard-to-undo operations, it is advisable not to use Cherry mode while in the OS kernel. Consequently, system calls and other entries to the kernel automatically exit Cherry mode.

However, Cherry blends well with context switching and multiprogramming. If a timer interrupt mandates a context switch for a process, the processor bails out of Cherry mode as described in Section 2.3.2, after which the resident process can be preempted safely. If it is the process who yields the CPU (e.g. due to a blocking semaphore), the system call itself exits Cherry mode, as described above. In no case is a rollback to the checkpoint necessary.

2.4 Optimal Size of a Cherry Cycle

The size of a Cherry cycle is crucial to the performance of Cherry. In what follows, we denote by T_c the duration of a Cherry cycle ignoring any Cherry overheads. For Cherry to work well, T_c has to be within a range. If T_c is too short, performance is hurt by the overhead of the checkpointing and the collapse step. If, instead, T_c is too long, both the probability of suffering an exception within a Cherry cycle and the cost of a rollback are high. The optimal T_c is found somewhere in between these two opposing conditions. We now show how to find the optimal T_c. For simplicity, in the following discussion we assume that the IPC in Cherry and non-Cherry mode stays constant at $s \cdot$IPC and IPC, respectively, where s denotes the average overhead-free speedup delivered by the Cherry mode.

We can express the per-Cherry-cycle overhead T_o of running in Cherry mode as:

$$T_o = c_k + p_e c_e \quad (1)$$

where c_k is the overhead caused by checkpointing and by the reduced performance experienced in the subsequent collapse step, p_e is the probability of suffering a rollback-causing exception in a Cherry cycle, and c_e is the cost of suffering such an exception. If exceptions occur every T_e cycles, with $T_c < T_e$, we can rewrite:

$$p_e = \frac{T_c}{T_e} \quad (2)$$

Notice that the expression for p_e is conservative, since it assumes that all exceptions cause rollbacks. In reality, only exceptions triggered by instructions in the irreversible set cause the processor to roll back, and thus the actual p_e would be lower.

To calculate the cost of suffering such an exception, we assume that when exceptions arrive, they do so half way into a Cherry cycle. In this case, the cost consists of re-executing half Cherry cycle at non-Cherry speed, plus the incremental overhead of executing (for the first time) another half Cherry cycle at non-Cherry speed rather than at Cherry speed. Recall that, after suffering an exception, we execute the instructions of one full Cherry cycle in non-Cherry mode (Section 2.3.3). Consequently:

$$c_e = s\frac{T_c}{2} + (s-1)\frac{T_c}{2} \quad (3)$$

The optimal T_c is the one that minimizes T_o/T_c. Substituting Equation 2 and Equation 3 in Equation 1, and dividing by T_c yields:

$$\frac{T_o}{T_c} = \frac{c_k}{T_c} + \left(s - \frac{1}{2}\right)\frac{T_c}{T_e} \quad (4)$$

This expression finds a minimum in:

$$T_c = \sqrt{\frac{c_k T_e}{s - \frac{1}{2}}} \quad (5)$$

Figure 3 plots the relative overhead T_o/T_c against the duration T_c of an overhead-free Cherry cycle (Equation 4). For that experiment, we borrow from our evaluation section (Section 5): 3.2GHz processor, $c_k = 18.75$ns (60 cycles), and $s = 1.06$. Then, we plot curves for duration of interval between exceptions T_e ranging from 200μs to 1000μs. The minimum in each curve yields the optimal T_c (Equation 5). As we can see from the figure, for the parameters assumed, the optimal T_c hovers around a few microseconds.

3 EARLY RESOURCE RECYCLING

To illustrate Cherry, we implement early recycling in the load/store unit (Section 3.1) and register file (Section 3.2). Early recycling could be applied to other resources as well.

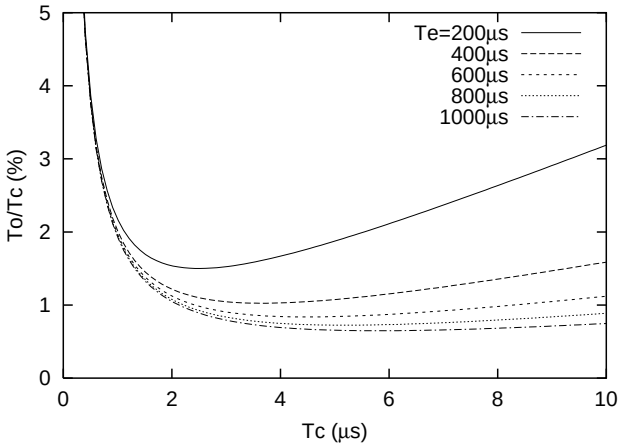


Figure 3: Example of Cherry overheads for different intervals between exceptions (T_e) and overhead-free Cherry cycle durations (T_c).

3.1 Load/Store Unit

Typical load/store units comprise one reorder queue for loads and one for stores [1, 21]. Either reorder queue may become a performance bottleneck if it fills up. In this section we first discuss a conventional design of the queues, and then we propose a new mechanism for early recycling of load/store queue entries.

3.1.1 Conventional Design

The processor assigns a Load Queue (LQ) entry to every load instruction in program order, as the instruction undergoes renaming. The entry initially contains the destination register. As the load executes, it fills its LQ entry with the appropriate physical address and issues the memory access. When the data are obtained from the memory system, they are passed to the destination register. Finally, when the load finishes and reaches the head of the ROB, the load instruction retires. At that point, the LQ entry is recycled.

Similarly, the processor assigns Store Queue (SQ) entries to every store instruction in program order at the renaming stage. As the store executes, it generates the physical address and the data value, which are stored in the corresponding SQ entry. An entry whose address and data are still unknown is said to be *empty*. When both address and data are known, and the corresponding store instruction reaches the head of the ROB, the update is sent to the memory system. At that point, the store retires and the SQ entry is recycled.

Address Disambiguation and Load-Load Replay

At the time a load generates its address, a disambiguation step is performed by comparing its physical address against that of older SQ entries. If a fully overlapping entry is found, and the data in the SQ entry are ready, the data are forwarded to the load directly. However, if the accesses fully overlap but the data are still missing, or if the accesses are only partially overlapping, the load is rejected, to be dispatched again after a number of cycles. Finally, if no overlapping store exists in the SQ that is older than the load, the load requests the data from memory at once.

The physical address is also compared against newer LQ entries. If an overlapping entry is found, the newer load and all its subsequent instructions are replayed, to eliminate the chance of an intervening store by another device causing an inconsistency.

This last event, called *load-load replay trap*, is meaningful only in environments where more than one device can be accessing the same memory region simultaneously, as in multiprocessors. In uniprocessor environments, such a situation could potentially be caused by processor and DMA accesses. However, in practice, it does not occur: the operating system ensures mutual exclusion of processor and DMA accesses by locking memory pages as needed. Consequently, load-load replay support is typically not necessary in uniprocessors.

Figure 4*(a)* shows an example of a load address disambiguation and a check for possible load-load replay traps.

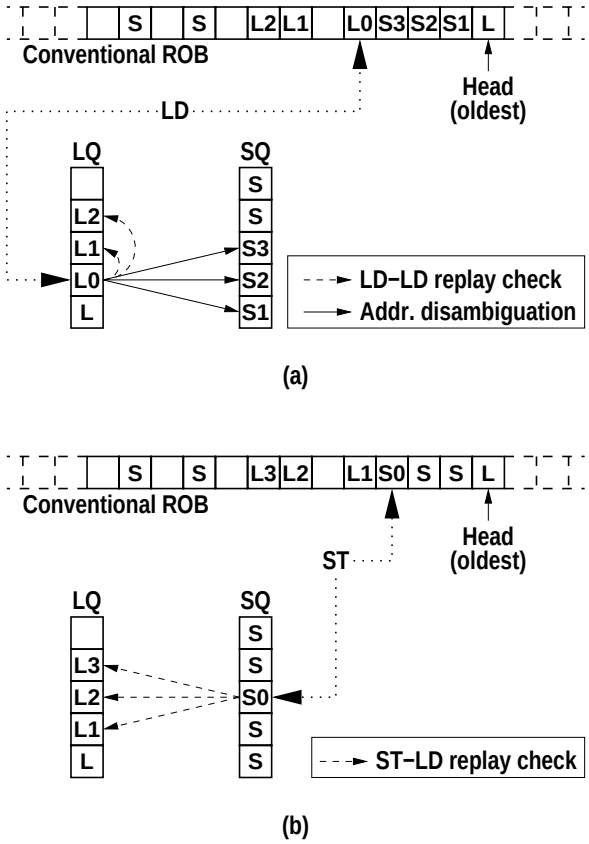


Figure 4: Actions taken on load *(a)* and store *(b)* operations in the conventional load/store unit assumed in this paper. L and Lx stand for Load, while S and Sx stand for Store.

Store-Load Replay

Once the physical address of a store is resolved, it is compared against newer entries in the LQ. The goal is to detect any *exposed* load, namely a newer load whose address overlaps with that of the store, without an intervening store that fully covers the load. Such a load has consumed data prematurely, either from memory or from an earlier store. Thus, the load and all instructions following it are aborted and replayed. This mechanism is called *store-load replay trap* [1].

Figure 4*(b)* shows an example of a check for possible store-load replay traps.

3.1.2 Design with Early Recycling

Following Cherry's philosophy, we want to release LQ and SQ entries as early as it is possible to do so. In this section, we describe the conditions under which this is the case.

Optimized LQ

As explained before, a LQ entry may trigger a replay trap if an older store (or older load in multiprocessors) resolves to an overlapping address. When we release a LQ entry, we lose the ability to compare against its address (Figure 4). Consequently, we can only release a LQ entry when such a comparison is no longer needed because no replay trap can be triggered.

To determine whether or not a LQ entry is needed to trigger a replay trap, we use the U_L and U_S pointers to the ROB (Section 2). Any load that is older than U_S cannot trigger a store-load replay trap, since the physical addresses of all older stores are already known. Furthermore, any load that is older than U_L cannot trigger a load-load replay trap, because the addresses of all older loads are already known.

In a typical uniprocessor environment, only store-load replay traps are relevant. Consequently, as the U_S moves in the ROB, any loads that are older than U_S release their LQ entry. In a multiprocessor or other multiple-master environment, both store-load and load-load replay traps are relevant. Therefore, as the U_S and U_L move in the ROB, any loads that are older than oldest(U_L, U_S) release their LQ entry.[2] To keep the LQ simple, LQ entries are released in order.

Early recycling of LQ entries is not limited by U_B; it is fine for load instructions whose entry has been recycled to be subject to branch mispredictions. Moreover, it is possible to service exceptions and interrupts inside the irreversible set without needing a rollback. This is because LQ entries that are no longer needed to detect possible replay traps can be safely recycled without creating a side effect. In light of an exception or interrupt, the recycling of such a LQ entry does not alter the processor state needed to service that exception or interrupt. Section 3.3 discusses how this blends in a Cherry implementation with recycling of other resources.

Since LQ entries are recycled early, we partition the LQ into two structures. The first one is called *Load Reorder Queue (LRQ)*, and supports the address checking functionality of the conventional LQ. Its entries are assigned in program order at renaming, and recycled according to the algorithm just described. Each entry contains the address of a load.

The second structure is called *Load Data Queue (LDQ)*, and supports the memory access functionality of the conventional LQ. LDQ entries are assigned as load instructions begin execution, and are recycled as soon as the data arrive from memory and are delivered to the appropriate destination register (which the entry points to). Because of their relatively short-lived nature, it is reasonable to assume that the LDQ does not become a bottleneck as we optimize the LRQ. LDQ entries are not assigned in program order, but the LDQ must be addressable by transaction id, so that the entry can be found when the data come back from memory.

Finally, note that, even for a load that has had its LRQ entry recycled, address checking proceeds as usual. Specifically, when its address is finally known, it is compared against older stores for possible data forwarding. This case only happens in uniprocessors, since in multiprocessors the PNR for LQ entries depends on oldest(U_L, U_S).

Optimized SQ

When we release a SQ entry, we must send the store to the memory system. Consequently, we can only release a SQ entry when the old value in the memory system is no longer needed. For the latter to be true, it is sufficient that: (1) no older load is pending address disambiguation, (2) no older load is subject to replay traps, and (3) the store is not subject to squash due to branch misprediction. Condition (1) means that all older loads have already generated their address and, therefore, located their "supplier", whether it is memory or an older store. If it is memory, recall that load requests are sent to memory as soon as their addresses are known. Therefore, if our store is older than U_L, it is guaranteed that all older loads that need to fetch their data from memory have already done so. Condition (2) implies that all older loads are older than oldest(U_S) (typical uniprocessor) or oldest(U_L, U_S) (multiprocessor or other multiple-master system), as discussed above. Finally, condition (3) implies that the store itself is older than U_B. Therefore, all conditions are met if the store itself is older than oldest(U_L, U_S, U_B).[3]

There are two additional implementation issues related to accesses to overlapping addresses. They are relevant when we send a store to the memory system and recycle its SQ entry. First, we would have to compare its address against all older entries in the SQ to ensure that stores to overlapping addresses are sent to the cache in program order. To simplify the hardware, we eliminate the need for such a comparison by simply sending the updates to the memory system (and recycling the SQ entries) in program order. In this case, in-order updates to overlapping addresses are automatically enforced.

Second, note that a store is not sent to the memory system until all previous loads have been resolved (store is older than U_L). One such load may be to an address that overlaps with that of the store. Recall that loads are sent to memory as soon as their addresses are known. The LRQ entry for the load may even be recycled. This case is perfectly safe if the cache system can ensure the ordering of the accesses. This can be implemented in a variety of ways (queue at the MSHR, reject store, etc.), whose detailed implementation is out of the scope of this work.

3.2 Register File

The register file may become a performance bottleneck if the processor runs out of physical registers. In this section, we briefly discuss a conventional design of the register file, and then propose a mechanism for early recycling of registers.

3.2.1 Conventional Design

In our design, we use a register map at the renaming stage of the pipeline and one at the retirement stage. At renaming, instructions pick one available physical register as the destination for their operation and update the renaming map accordingly. Similarly, when the instruction retires, it updates the retirement map to reflect the architectural state immediately after the instruction. Typically, the retirement map is used to support precise exception handling: when an instruction raises an exception, the processor waits until such instruction reaches the ROB head, at which point the processor has a precise image of the architectural state before the exception.

Physical registers holding architectural values are recycled when a retiring instruction updates the retirement map to point away from them. Thus, once a physical register is allocated *at renaming* for an instruction, it remains "pinned" until a subsequent instruction supersedes it *at retirement*. However, a register may become *dead* much earlier: as soon as it is superseded *at renaming*, and all its consumer instructions have read its value. From this moment, and until the superseding instruction retires, the register remains pinned in case the superseding instruction is rolled back for whatever reason, e.g. due to branch misprediction or exception. This effect causes a suboptimal utilization of the register file.

[2] Note that the LQ entry for the load *equal* to oldest(U_L, U_S) cannot trigger a replay trap and, therefore, can also be released. However, for simplicity, we ignore this case.

[3] Note that it is safe to update the memory system if the store is *equal* to oldest(U_L, U_S, U_B). However, for simplicity, we ignore this case.

3.2.2 Design with Early Recycling

Following Cherry's philosophy, we recycle dead registers as soon as possible, so that they can be reallocated by new instructions. However, we again need to rely on checkpointing to revert to a correct state in case of an exception in an instruction in the irreversible set.

We recycle a register when the following two conditions hold. First, the instruction that produces the register and all those that consume it must be (1) executed and (2) both free of replay traps and not subject to branch mispredictions. The latter implies that they are older than oldest(U_S, U_B) (typical uniprocessor) or oldest(U_L, U_S, U_B) (multiprocessor or other multiple-master system), as discussed above.

The second condition is that the instruction that supersedes the register is not subject to branch mispredictions (older than U_B). Squashing such an instruction due to a branch misprediction would have the undesirable effect of reviving the superseded register. Notice, however, that the instruction can harmlessly be re-executed due to a memory replay trap, and thus ordering constraints around $\{U_L, U_S\}$ are unnecessary. In practice, to simplify the implementation, we also require that the instruction that supersedes the register be older than oldest(U_S, U_B) (typical uniprocessor) or oldest(U_L, U_S, U_B) (multiprocessor or other multiple-master system).

In our implementation, we augment every physical register with a *Superseded* bit and a *Pending* count. This support is similar to [17]. The Superseded bit marks whether the instruction that supersedes the register is older than oldest(U_S, U_B) (or oldest(U_L, U_S, U_B) in multiprocessors), which implies that so are all consumers. The Pending count records how many instructions among the consumers and producer of this register are older than oldest(U_S, U_B) (or oldest(U_L, U_S, U_B) in multiprocessors) and have not yet completed execution. A physical register can be recycled only when the Superseded bit is set and the Pending count is zero. Finally, we also assume that instructions in the ROB keep, as part of their state, a pointer to the physical register that their execution supersedes. This support exists in the MIPS R10000 processor [23].

As an instruction goes past oldest(U_S, U_B) (or oldest(U_L, U_S, U_B) in multiprocessors), the proposed new bits in the register file are acted upon as follows: (1) If the instruction has not finished execution, the Pending count of every source and destination register is incremented; (2) irrespective of whether the instruction has finished execution, the Superseded bit of the superseded register, if any, is set; (3) if the superseded register has both a set Superseded bit and a zero Pending count, the register is added to the free list.

Additionally, every time that an instruction past oldest(U_S, U_B) (or oldest(U_L, U_S, U_B) in multiprocessors), finishes executing, it decrements the Pending count of its source and destination registers. If the Pending count of a register reaches zero and its Superseded bit is set, that register is added to the free list.

Overall, in Cherry mode, register recycling occurs before the retirement stage. Note that, upon a collapse step, the processor seamlessly switches from Cherry to non-Cherry register recycling. This is because, at the time the irreversible set is fully collapsed, all early recycled registers in Cherry (and only those) would have also been recycled in non-Cherry mode.

3.3 Putting It All Together

In this section we have applied Cherry's early recycling approach to three different types of resources: LQ entries, SQ entries, and registers. When considered separately, each resource defines its own PNR and irreversible set. Table 1 shows the PNR for each of these three resources.

When combining early recycling of several resources, we define the dominating PNR as the one which is *farthest* from the ROB head at each point in time. Exceptions on instructions older than that PNR typically require a rollback to the checkpoint; exceptions on instructions newer than that PNR can simply trigger a collapse step so that the processor falls back to non-Cherry mode.

Resource	PNR Value
LQ entries (uniprocessor)	U_S
LQ entries (multiprocessor)	oldest(U_L, U_S)
SQ entries	oldest(U_L, U_S, U_B)
Registers (uniprocessor)	oldest(U_S, U_B)
Registers (multiprocessor)	oldest(U_L, U_S, U_B)

Table 1: PNR for each of the example resources that are recycled early under Cherry.

However, it is important to note that our proposal for early recycling of LQ entries is a special case: it guarantees precise handling of extraordinary events even when they occur within the irreversible set (Section 3.1.2). As a result, the PNR for LQ entries need not be taken into account when determining the dominating PNR. Thus, for a Cherry processor with recycling at these three points, the dominating PNR is the newest of the PNRs for SQ entries and for registers. In a collapse step, the dominating PNR is the one that freezes until the ROB head catches up with it.

4 EVALUATION SETUP

Simulated Architecture

We evaluate Cherry using execution-driven simulations with a detailed model of a state-of-the-art processor and its memory subsystem. The baseline processor modeled is an eight-issue dynamic superscalar running at 3.2GHz that has two levels of on-chip caches. The details of the *Baseline* architecture modeled are shown in Table 2. In our simulations, the latency and occupancy of the structures in the processor pipeline, caches, bus, and memory are modeled in detail.

Processor			
Frequency: 3.2GHz			Branch penalty: 7 cycles (minimum)
Fetch/issue/commit width: 8/8/12			Up to 1 taken branch/cycle
I. window/ROB size: 128/384			RAS: 32 entries
Int/FP registers : 192/128			BTB: 4K entries, 4-way assoc.
Ld/St units: 2/2			Branch predictor:
Int/FP/branch units: 7/5/3			Hybrid with speculative update
Ld/St queue entries: 32/32			Bimodal size: 8K entries
MSHRs: 24			Two-level size: 64K entries
Cache	L1	L2	Bus & Memory
Size:	32KB	512KB	FSB frequency: 400MHz
RT:	2 cycles	10 cycles	FSB width: 128bit
Assoc:	4-way	8-way	Memory: 4-channel Rambus
Line size:	64B	128B	DRAM bandwidth: 6.4GB/s
Ports:	4	1	Memory RT: 120ns

Table 2: *Baseline* architecture modeled. In the table, MSHR, RAS, FSB and RT stand for Miss Status Handling Register, Return Address Stack, Front-Side Bus, and Round-Trip time from the processor, respectively. Cycle counts refer to processor cycles.

The processor has separate structures for the ROB, instruction window, and register file. When an instruction is issued, it is placed in both the instruction window and the ROB. Later, when all the input operands are available, the instruction is dispatched to the functional units and is removed from the instruction window.

In our simulations, we break down the execution time based on the reason why, for each issue slot in each cycle, the opportunity to insert a useful instruction into the instruction window is missed (or not). If, for a particular issue slot, an instruction is inserted into the instruction window, and that instruction eventually graduates, that slot is counted as busy. If, instead, an instruction is available but is not inserted in the instruction window because a necessary resource is unavailable, the missed opportunity is attributed to such a resource. Example of such resources are load queue entry, store queue entry, register, or instruction window entry. Finally, instructions from mispredicted paths and other overheads are accounted for separately.

We also simulate four enhanced configurations of the *Baseline* architecture: *Base2*, *Base3*, *Base4*, and *Limit*. Going from *Baseline* to *Base2*, we simply add 32 load queue entries, 32 store queue entries, 32 integer registers, and 32 FP registers. The same occurs as we go from *Base2* to *Base3*, and from *Base3* to *Base4*. *Limit* has an unlimited number of load/store queue entries and integer/FP registers.

Cherry Architecture

We simulate the *Baseline* processor with Cherry support (*Cherry*). We estimate the cost of checkpointing the architectural registers to be 8 cycles. Moreover, we use simulations to derive an average overhead of 52 cycles for a collapse step. Consequently, c_k becomes 60 cycles. If we set the duration of an overhead-free Cherry cycle (T_c) to 5μs, the c_k overhead becomes negligible. Under these conditions, equation 4 yields a total relative overhead (T_o/T_c) of at most one percent, if the separation between exceptions (T_e) is 448μs or more. Note that, in equation 4, we use an average overhead-free Cherry speedup (s) of 1.06. This number is what we obtain for SPECint applications in Section 5. In our evaluation, however, we do not model exceptions. Neglecting them does not introduce significant inaccuracy, given that we simulate applications in steady state, where page faults are infrequent.

Applications

We evaluate Cherry using most of the applications of the SPEC CPU2000 suite [5]. The first column of Table 3 in Section 5.1 lists these. Some applications from the suite are missing; this is due to limitations in our simulation infrastructure. For these applications, it is generally too time-consuming to simulate the reference input set to completion. Consequently, in all applications, we skip the initialization, and then simulate 750 million instructions. If we cannot identify the initialization code, we skip the first 500 million instructions before collecting statistics. The applications are compiled with -O2 using the native SGI MIPSPro compiler.

5 EVALUATION

5.1 Overall Performance

Figures 5 and 6 show the speedups obtained by the Cherry, Base2, Base3, Base4, and Limit configurations over the Baseline system. The figures correspond to the SPECint and SPECfp applications, respectively, that we study. For each application, we show two bars. The leftmost one (*R*) uses the realistic branch prediction scheme of Table 2. The rightmost one (*P*) uses perfect branch prediction for both the advanced and Baseline systems. Note that, even is this case, Cherry uses U_B.

The figures show that Cherry yields speedups across most of the applications. The speedups are more modest in SPECint applications, where Cherry's average performance is between that of Base2 and Base3. For SPECfp applications, the speedups are higher. In this case, the average performance of Cherry is close to that of Base4 and Limit. Overall, with the realistic branch prediction, the average speedup of Cherry on SPECint and SPECfp applications is 1.06 and 1.26, respectively.

If we compare the bars with realistic and perfect branch prediction, we see that some SPECint applications experience significantly higher speedups when branch prediction is perfect. This is the case for both Cherry and enhanced non-Cherry configurations. The reason is that an increase in available resources through early recycling (Cherry) or by simply adding more resources (Base2 to Base4 and Limit) increases performance when these resources are *successfully* re-utilized by instructions that would otherwise wait. Thus, if branch prediction is poor, most of these extra resources are in fact wasted by speculative instructions whose execution is ultimately moot. In perlbmk, for example, the higher speedups attained when all configurations (including Baseline) operate with perfect branch prediction is due to better resource utilization. On the other hand, SPECfp applications are largely insensitive to this effect, since branch prediction is already very successful in the realistic setup.

In general, the gains of Cherry come from recycling resources. To understand the degree of recycling, Table 3 characterizes the irreversible set and other related Cherry parameters. The data corresponds to realistic branch prediction. Specifically, the second column shows the average fraction of ROB entries that are used. The next three columns show the size of the irreversible set, given as a fraction of the used ROB. Recall that the irreversible set is the distance between the PNR and the ROB head (Figure 1). Since the irreversible set depends on the resource being recycled, we give separate numbers for register, LQ entry, and SQ entry recycling. As indicated in Section 3.3, the PNR in uniprocessors is oldest(U_S, U_B) for registers, U_S for LQ entries, and oldest(U_L, U_S, U_B) for SQ entries. Finally, the last column shows the average duration of the collapse step. Recall from Section 3.3 that it involves identifying the newest of the PNR for registers and for SQ entries, and freezing it until the ROB head catches up with it.

	Apps	Used ROB (%)	Irreversible Set (% of Used ROB)			Collapse Step (Cycles)
			Reg	LQ	SQ	
SPECint	bzip2	29.9	24.3	55.8	19.5	292.3
	crafty	28.8	33.4	97.6	28.6	41.9
	gcc	19.1	19.0	82.3	17.8	66.9
	gzip	28.5	65.5	81.7	8.5	47.1
	mcf	30.1	14.6	37.7	13.8	695.6
	parser	30.7	26.1	80.7	21.8	109.2
	perlbmk	12.2	24.6	89.9	20.5	23.3
	vortex	39.3	26.3	87.1	24.9	64.4
	vpr	32.9	25.2	83.6	21.5	165.1
	Average	**27.9**	**28.7**	**77.4**	**19.7**	**167.3**
SPECfp	applu	62.2	61.6	62.4	60.7	411.5
	apsi	76.8	82.3	83.1	81.6	921.1
	art	88.0	54.3	62.6	29.2	1247.3
	equake	41.6	61.6	69.1	57.3	135.2
	mesa	29.8	35.1	44.6	34.6	33.7
	mgrid	65.1	91.5	93.5	91.3	335.9
	swim	59.4	64.8	65.4	64.7	949.1
	wupwise	71.9	90.3	71.2	87.9	190.7
	Average	**61.9**	**67.7**	**78.3**	**63.4**	**528.1**

Table 3: Characterizing the irreversible set and other related Cherry parameters.

Consider the SPECint applications first. The irreversible set for the LQ entries is very large. Its average size is about 77% of the used ROB. This shows that U_S moves far ahead of the ROB head. On the other hand, the irreversible set for the registers is much smaller. Its average size is about 29% of the used ROB. This means that oldest(U_S, U_B) is not far from the ROB head. Consequently, U_B

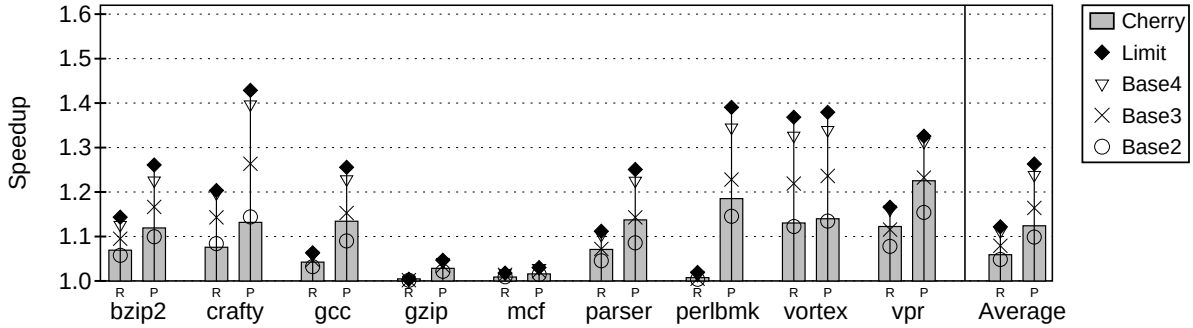


Figure 5: Speedups delivered by the Cherry, Base2, Base3, Base4, and Limit configurations over the Baseline system, for the SPECint applications that we study. For each application, the R and P bars correspond to realistic and perfect branch prediction, respectively.

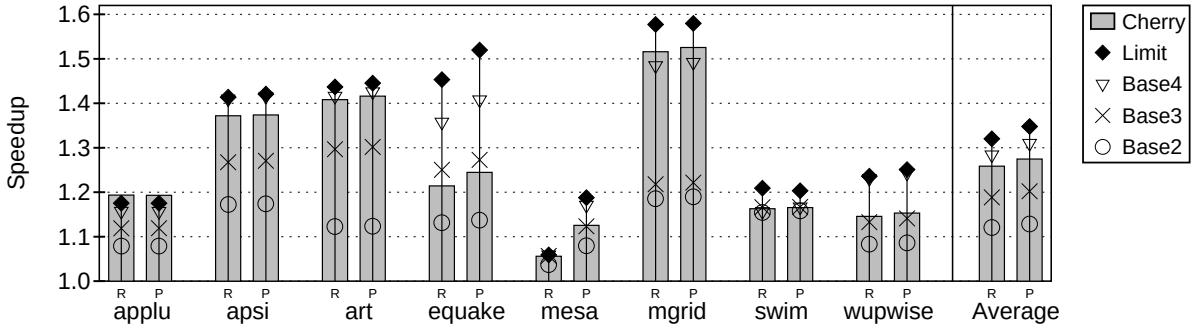


Figure 6: Speedups delivered by the Cherry, Base2, Base3, Base4, and Limit configurations over the Baseline system, for the SPECfp applications that we study. For each application, the R and P bars correspond to realistic and perfect branch prediction, respectively.

is the pointer that keeps the PNR from advancing. In these applications, branch conditions often depend on long-latency instructions and, as a result, they remain unresolved for a while. Finally, the irreversible set for the SQ entries is even smaller. Its average size is about 20%. In this case, PNR is given by oldest(U_L, U_S, U_B) and it shows that U_L further slows down the move of the PNR. In these applications, load addresses often depend on long-latency instructions too.

In contrast, SPECfp applications have fewer conditional branches and they are resolved faster. Furthermore, load addresses follow a more regular pattern and are also resolved earlier. As a consequence, the PNRs for register and SQ entries move far ahead of the ROB head. The result is that Cherry delivers a much higher speedup for SPECfp applications (Figure 6) that for SPECint (Figure 5).

We note that the larger irreversible sets for the SPECfp applications imply a higher cost for the collapse step. Specifically, the average collapse step goes from 167 to 528 cycles as we go from SPECint to SPECfp applications. A long collapse step increases the term c_k in Equation 4, which forces T_c to be longer.

5.2 Contribution of Resources

Figures 7 and 8 show the contribution of different components to the execution time of the SPECint and SPECfp applications, respectively. Each application shows the execution time for three configurations, namely Baseline, Cherry, and Limit. The execution times are normalized to Baseline. The bars are broken down into busy time (*Busy*) and different types of processor stalls due to: lack of physical registers (*Regs*), lack of SQ entries (*SQ*), lack of load queue entries (*LQ*). A final category (*Other*) includes other losses, including those due to branch mispredictions or lack of entries in the instruction window. Section 4 discussed how we obtain these categories.

The Baseline bars show that of the three potential bottlenecks targeted in this paper, the LQ is by far the most serious one. Lack of LQ entries causes a large stall in SPECint and, especially, SPECfp applications.

Our proposal of early recycling of LQ entries is effective in both the SPECint and SPECfp applications. Our optimization reduces most of the LQ stall. It unleashes extra ILP, which in turn puts more pressure on the SQ, register file, and other resources. Even though Cherry does recycle some SQ entries and physical registers, the net effect of our oprimizations is an increased level of saturation on these two resources for both SPECint and SPECfp applications.

One reason why Cherry is not as effective in recycling SQ entries and registers is that their PNRs are constrained by more conditions. Indeed, the PNR for registers is oldest(U_S, U_B), while the one for SQ entries is oldest(U_L, U_S, U_B). In particular, U_B limits the impact of Cherry noticeably.

Overall, to enhance the impact of Cherry, we can improve in two different ways. First, we can design techniques to advance the PNR for SQ entries and registers more aggressively. However, this may increase the risk of a rollback. Second, recycling within the current irreversible set can be done more aggressively. This adds complexity, and may also increase the risk of rollbacks.

5.3 Resource Utilization

To gain a better insight into the performance results of Cherry, we measure the usage of each of the targeted resources. Figure 9 shows cumulative distributions of usage for each of the resources. From top to bottom, the charts refer to LQ entries, SQ entries, integer registers, and floating-point registers. In each chart, the horizontal axis is

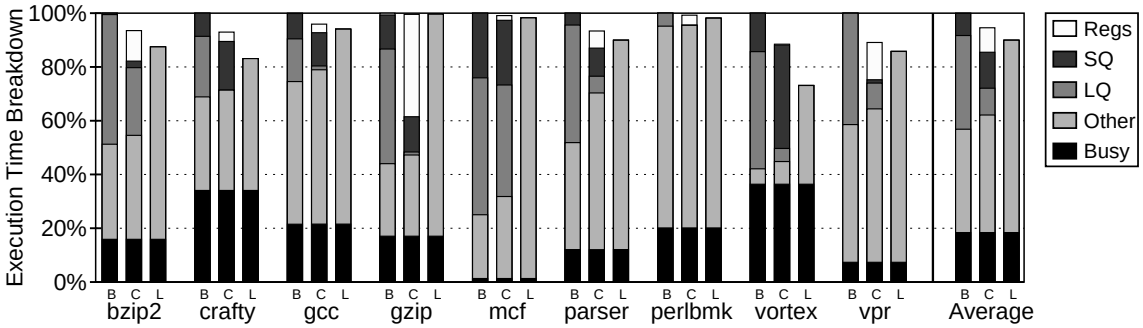


Figure 7: Breakdown of the execution time of the SPECint applications for the Baseline (B), Cherry (C), and Limit (L) configurations.

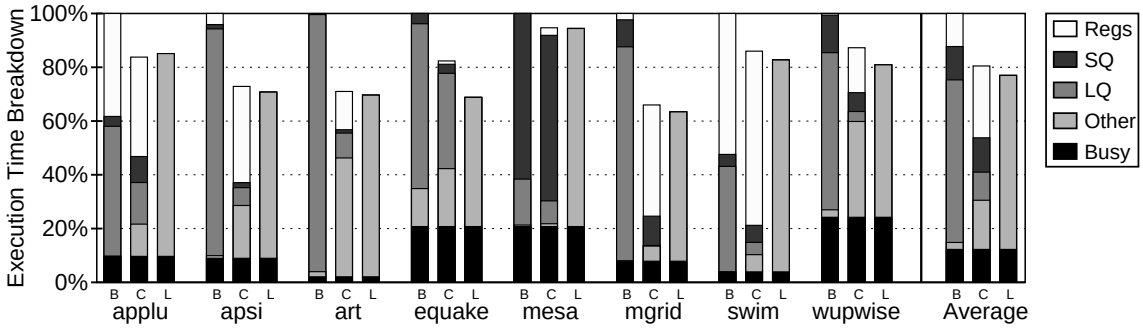


Figure 8: Breakdown of the execution time of the SPECfp applications for the Baseline (B), Cherry (C), and Limit (L) configurations.

the cumulative percentage of time that a resource is allocated below the level shown in the vertical axis. Each chart shows the distribution for the Baseline, Limit, and two Cherry configurations. The latter correspond to the *real* number of allocated entries (*CherryR*) and the *effective* number of allocated entries (*CherryE*). The effective entries include both the entries that are actually occupied and those that would have been occupied had they not been recycled. The difference between CherryE and CherryR shows how effective Cherry is in recycling a given resource. Finally, the area under each curve is proportional to the average usage of a resource.

The top row of Figure 9 shows that LQ entry recycling is very effective. Under Baseline, the LQ is full about 45% and 65% of the time in SPECint and SPECfp applications, respectively. With Cherry, in more than half of the time, all the LQ entries are recycled. We see that the LQ is almost full less than 15% of the time. Moreover, the effective number of LQ entries is significantly larger than the actual size of the LQ.

The second row of Figure 9 shows that, as expected, the recycling of SQ entries is less effective. In SPECint applications, the effective size of the SQ under Cherry surpasses the actual size of that resource significantly in only 6% of the time. However, the potential demand for SQ entries (in the Limit configuration) is much larger. The situation in SPECfp applications is slightly different. The SQ entries are recycled somewhat more effectively.

The last two rows of Figure 9 show the usage of integer (third row) and floating-point (bottom row) registers. In the SPECint applications, the recycling of registers is not very effective. The reason for this is the same as for SQ entries: the PNR is unable to sufficiently advance to permit effective recycling. In contrast, the PNR advances quite effectively in SPECfp applications. The resulting degree of register recycling is very good. Indeed, the effective number of integer registers approaches the potential demand. The potential demand for floating-point registers is larger and is difficult to meet. However, the effective number of floating-point registers in Cherry is larger than the actual size of the register file 50% of the time. In particular, it is more than twice the actual size of the register file 15% of the time.

6 COMBINING CHERRY AND SPECULATIVE MULTITHREADING

6.1 Similarities and Differences

Speculative multithreading (SM) is a technique where several tasks are extracted from a sequential code and executed speculatively in parallel [4, 9, 14, 19, 20]. Value updates by speculative threads are buffered, typically in caches. If a cross-thread dependence violation is detected, updates are discarded and the speculative thread is rolled back to a safe state. The existence of at least one safe thread at all times guarantees forward progress. As safe threads finish execution, they propagate their nonspeculative status to successor threads.

Cherry and SM are complementary techniques: while Cherry uses potentially unsafe resource recycling to enhance instruction overlap *within* a thread, SM uses potentially unsafe parallel execution to enhance instruction overlap *across* threads. Furthermore, Cherry and SM share much of their hardware requirements. Consequently, combining these two schemes becomes an interesting option.

Cherry and SM share two important primitives. The first one is support to checkpoint the processor's architectural state before entering unsafe execution, and to roll back to it if the program state becomes inconsistent. The second primitive consists of support to buffer unsafe memory state in the caches, and either merge it with the memory state when validated, or invalidate it if proven corrupted.

Naturally, both SM and Cherry have additional requirements of their own. SM often tags cached data and accesses with a thread

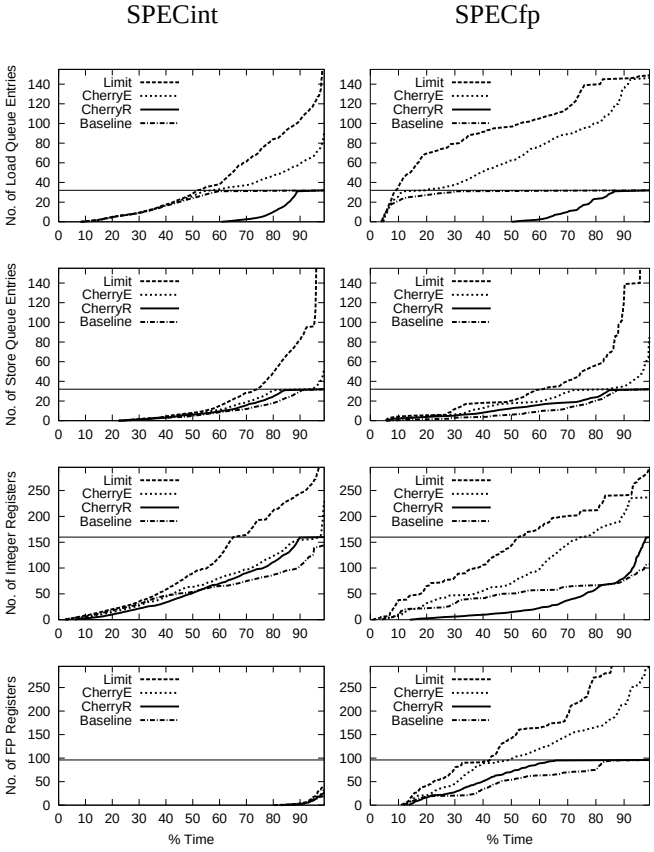


Figure 9: Cumulative distribution of resource usage in SPECint (left) and SPECfp (right) applications. The horizontal axis is the cumulative percentage of time that a resource is used below the level shown in the vertical axis. The resources are, from top to bottom: LQ entries, SQ entries, integer physical registers, and floating-point physical registers.

ID, which identifies the owner or originator thread. Furthermore, SM needs hardware or software to check for cross-thread dependence violations. On the other hand, Cherry needs support to recycle load/store queue entries and registers, and to maintain the PNR pointer.

6.2 Combined Scheme

In a processor that supports both SM and Cherry execution, we propose to exploit both schemes by enabling/disabling speculative execution and Cherry mode in lockstep. Specifically, as a thread becomes speculative, it also enters Cherry mode, and when it successfully completes the speculative section, it also completes the Cherry cycle. Moreover, if speculation is aborted, so is the Cherry cycle, and vice versa. We now show that this approach has the advantage of reusing hardware support.

Enabling and disabling the two schemes in lockstep reuses the checkpoint and the cache support. Indeed, a single checkpoint is required when the thread enters both speculative execution and Cherry mode at once. As for cache support, SM typically tags each cache line with a Read and Write bit which, roughly speaking, are set when the speculative thread reads or writes the line, respectively. On the other hand, Cherry tags cache lines with the Volatile bit, which is set when the thread writes the line. Consequently, the Write and Volatile bits can be combined into one.

With such support, when the thread is speculative, any write sets the Write/Volatile bit. When the thread becomes nonspeculative, all Read bits in the cache are gang-cleared. Then, the processor exits Cherry mode and also gang-clears all Write/Volatile bits.

Special consideration has to be given to cache overflow situations. Under SM alone, the speculative thread stalls when the cache is about to overflow. Under Cherry mode alone, an interrupt informs the processor when the number of Volatile lines in the victim cache exceeds a certain threshold. This advance notice allows the processor to return to non-Cherry mode immediately without overflowing the cache. When we combine both SM and Cherry mode, we stall the processor as soon as the advance notice is received. When the thread later becomes nonspeculative, the thread can resume and immediately return to non-Cherry. Thanks to stalling when the advance notice was received, there is still some room in the cache for the thread to complete the Cherry cycle and not overflow. This strategy is likely to avoid an expensive rollback to the checkpoint.

Another consideration related to the previous one is the treatment of the advance warning interrupt when combining Cherry and SM. Note that the advance notice interrupt in Cherry requires no special handling. Indeed, *any* interrupt triggers the ending of the current Cherry—the advance warning interrupt is special only in that it is signaled when the cache is nearly full. However, when Cherry and SM are combined, the advance warning interrupt has to be recognized as such, so that the stall can be performed before the processor's interrupt handling logic can react to it. This differs from the way other interrupts are handled in SM, where interrupts are typically handled by squashing the speculative thread and responding to the interrupt immediately.

7 RELATED WORK

Our work combines register checkpointing and reorder buffer (ROB) to allow precise exceptions, fast handling of frequent instruction replay events, and recycling of load and store queue entries and registers. Previous related work can be divided into the following four categories.

The first category includes work on precise exception handling. Hwu and Patt [7] use checkpointing to support precise exceptions in out-of-order processors. On an exception, the processor rolls back to the checkpoint, and then executes code *in order* until the excepting instruction is met. Smith and Pleszkun [18] discuss several methods to support precise exceptions. The Reorder Buffer (ROB) and the History Buffer are presented, among other techniques.

The second category includes work related to register recycling. Moudgill et al. [17] discuss performing early register recycling in out-of-order processors that support precise exceptions. However, the implementation of precise exceptions in [17] relies on either checkpoint/rollback for every replay event, or a history buffer that restricts register recycling to only the instruction at the head of that buffer. In contrast, Cherry combines the ROB and checkpointing, allowing register recycling and, at the same time, quick recovery from frequent replay events using the ROB, and precise exception handling using checkpointing. Wallace and Bagherzadeh [22], and later Monreal et al. [16] delay allocation of physical registers to the execution stage. This is complementary to our work, and can be combined with it to achieve even better resource utilization. Lozano and Gao [12], Martin et al. [15], and Lo et al. [11] use the compiler to analyze the code and pass on dead register information to the hardware, in order to deallocate physical registers. The latter approaches require instruction set support: special symbolic registers [12], register kill instructions [11, 15], or cloned versions of opcodes that implicitly kill registers [11]. Our approach does not require changes in the instruction set or compiler support; thus, it works with legacy application binaries.

The third category of related work would include work that recycles load and store queue entries. Many current processors support speculative loads and replay traps [1, 21] and, to the best of our knowledge, this is the first proposal for early recycling of load and store queue entries in such a scenario.

The last category includes work that, instead of recycling resources early to improve utilization, opts to build larger structures for these resources. Lebeck et al. [10] propose a two-level hierarchical instruction window to keep the effective sizes large and yet the primary structure small and fast. The buffering of the state of all the in-flight instructions is achieved through the use of two-level register files similar to [3, 24], and a large load/store queue. Instead, we focus on improving the effective size of resources while keeping their actual sizes small. We believe that these two techniques are complementary, and could have an additive effect.

Finally, we notice that, concurrently to our work, Cristal et al. [2] propose the use of checkpointing to allow early release of unfinished instructions from the ROB and subsequent out-of-order commit of such instructions. They also leverage this checkpointing support to enable early register release. As a result, a large *virtual* ROB that tolerates long-latency operations can be constructed from a small physical ROB. This technique is compatible with Cherry, and both schemes could be combined for greater overall performance.

8 SUMMARY AND CONCLUSIONS

This paper has presented *CHeckpointed Early Resource RecYcling* (*Cherry*), a mode of execution that decouples the recycling of the resources used by an instruction and the retirement of the instruction. Resources are recycled early, resulting in a more efficient utilization. Cherry relies on state checkpointing to service exceptions for instructions whose resources have been recycled. Cherry leverages the ROB to (1) not require in-order execution as a fallback mechanism, (2) allow memory replay traps and branch mispredictions without rolling back to the Cherry checkpoint, and (3) quickly fall back to conventional out-of-order execution without rolling back to the checkpoint or flushing the pipeline. Furthermore, Cherry enables long checkpointing intervals by allowing speculative updates to reside in the local cache hierarchy.

We have presented a Cherry implementation that targets three resources: load queue, store queue, and register files. We use simple rules for recycling these resources. We report average speedups of 1.06 and 1.26 on SPECint and SPECfp applications, respectively, relative to an aggressive conventional architecture. Of the three techniques, our proposal for load queue entry recycling is the most effective one, particularly for integer codes.

Finally, we have described how to combine Cherry and speculative multithreading. These techniques complement each other and can share significant hardware support.

ACKNOWLEDGMENTS

The authors would like to thank Rajit Manohar, Sanjay Patel, and the anonymous reviewers for useful feedback.

REFERENCES

[1] Compaq Computer Corporation. *Alpha 21264/EV67 Microprocessor Hardware Reference Manual*, Shrewsbury, MA, September 2000.

[2] A. Cristal, M. Valero, J.-L. Llosa, and A. González. Large virtual ROBs by processor checkpointing. Technical Report UPC-DAC-2002-39, Universitat Politècnica de Catalunya, July 2002.

[3] J. L. Cruz, A. González, M. Valero, and N. P. Topham. Multiple-banked register file architectures. In *International Symposium on Computer Architecture*, pages 316–325, Vancouver, Canada, June 2000.

[4] L. Hammond, M. Wiley, and K. Olukotun. Data speculation support for a chip multiprocessor. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 58–69, San Jose, CA, October 1998.

[5] J. L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *IEEE Computer*, 33(7):28–35, July 2000.

[6] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal*, Q1 2001.

[7] W. W. Hwu and Y. N. Patt. Checkpoint repair for out-of-order execution machines. In *International Symposium on Computer Architecture*, pages 18–26, Pittsburgh, PA, June 1987.

[8] A. KleinOsowski, J. Flynn, N. Meares, and D. Lilja. Adapting the SPEC 2000 benchmark suite for simulation-based computer architecture research. In *Workshop on Workload Characterization*, Austin, TX, September 2000.

[9] V. Krishnan and J. Torrellas. A chip-multiprocessor architecture with speculative multithreading. *IEEE Transactions on Computers*, 48(9):866–880, September 1999.

[10] A. R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg. A large, fast instruction window for tolerating cache misses. In *International Symposium on Computer Architecture*, pages 59–70, Anchorage, AK, May 2002.

[11] J. L. Lo, S. S. Parekh, S. J. Eggers, H. M. Levy, and D. M. Tullsen. Software-directed register deallocation for simultaneous multithreaded processors. *IEEE Transactions on Parallel and Distributed Systems*, 10(9):922–933, September 1999.

[12] L. A. Lozano and G. R. Gao. Exploiting short-lived variables in superscalar processors. In *International Symposium on Microarchitecture*, pages 293–302, Ann Arbor, MI, November–December 1995.

[13] R. Manohar. Personal communication, August 2002.

[14] P. Marcuello and A. González. Clustered speculative multithreaded processors. In *International Conference on Supercomputing*, pages 365–372, Rhodes, Greece, June 1999.

[15] M. M. Martin, A. Roth, and C. N. Fischer. Exploiting dead value information. In *International Symposium on Microarchitecture*, pages 125–135, Research Triangle Park, NC, December 1997.

[16] T. Monreal, A. González, M. Valero, J. González, and V. Viñals. Delaying physical register allocation through virtual-physical registers. In *International Symposium on Microarchitecture*, pages 186–192, Haifa, Israel, November 1999.

[17] M. Moudgill, K. Pingali, and S. Vassiliadis. Register renaming and dynamic speculation: An alternative approach. In *International Symposium on Microarchitecture*, pages 202–213, Austin, TX, December 1993.

[18] J. E. Smith and A. R. Pleszkun. Implementing precise interrupts in pipelined processors. *IEEE Transactions on Computers*, 37(5):562–573, May 1988.

[19] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *International Symposium on Computer Architecture*, pages 414–425, Santa Margherita Ligure, Italy, June 1995.

[20] J. G. Steffan and T. C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *International Symposium on High-Performance Computer Architecture*, pages 2–13, Las Vegas, NV, January–February 1998.

[21] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–25, January 2002.

[22] S. Wallace and N. Bagherzadeh. A scalable register file architecture for dynamically scheduled processors. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 179–184, Boston, MA, October 1996.

[23] K. C. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, 6(2):28–40, April 1996.

[24] J. Zalamea, J. Llosa, E. Ayguadé, and M. Valero. Two-level hierarchical register file organization for VLIW processors. In *International Symposium on Microarchitecture*, pages 137–146, Monterey, CA, December 2000.

Positional Adaptation of Processors: Application to Energy Reduction *

Michael C. Huang
Dept. of Electrical and Computer Engineering
University of Rochester
michael.huang@ece.rochester.edu

Jose Renau and **Josep Torrellas**
Dept. of Computer Science
University of Illinois at Urbana-Champaign
{renau,torrellas}@cs.uiuc.edu

Abstract

Although adaptive processors can exploit application variability to improve performance or save energy, effectively managing their adaptivity is challenging. To address this problem, we introduce a new approach to adaptivity: the *Positional* approach. In this approach, both the *testing* of configurations and the *application* of the chosen configurations are associated with particular code sections. This is in contrast to the currently-used *Temporal* approach to adaptation, where both the testing and application of configurations are tied to successive intervals in *time*.

We propose to use *subroutines* as the granularity of code sections in positional adaptation. Moreover, we design three implementations of subroutine-based positional adaptation that target energy reduction in three different workload environments: embedded or specialized server, general purpose, and highly dynamic. All three implementations of positional adaptation are much more effective than temporal schemes. On average, they boost the energy savings of applications by 50% and 84% over temporal schemes in two experiments.

1 Introduction

Processor adaptation offers a major opportunity to the designers of modern processors. Currently, many proposed architectural enhancements have the desired effect (e.g. improve performance or save energy) *on average* for the whole program, but have the opposite effect during some periods of program execution. If the processor were able to adapt as the application executes by dynamically activating/deactivating the enhancement, the average performance improvement or energy savings would be higher.

Perhaps the area where adaptive processors have been studied the most is the low-power domain — this is why we focus the analysis in this paper on this area. In this case, researchers have proposed various architectural Low-Power Techniques (LPTs) that allow general-purpose processors to save energy, typically at the expense of performance (e.g. [1, 2, 4, 8, 17, 19]). Examples of such LPTs are cache reconfiguration and issue-width changes. By activating these LPTs dynamically, processors can be more effective. Some of the more advanced proposals for adaptive processors combine several LPTs [7, 12, 13, 15, 21].

*This work was supported in part by the National Science Foundation under grants EIA-0081307, EIA-0072102, and CHE-0121357; by DARPA under grant F30602-01-C-0078; and by gifts from IBM and Intel.

Unfortunately, controlling processor adaptation effectively is challenging. Indeed, an adaptive processor with multiple LPTs needs to make the twin decisions of when to adapt the hardware and what specific LPTs to activate. These decisions are usually based on testing a few different configurations of the LPTs and identifying which ones are best, and when.

Nearly all existing proposals for adaptive systems follow what we call a *Temporal* approach to adaptation [1, 2, 4, 6, 7, 8, 9, 12, 15, 19, 21]. In this case, both the testing (or exploration) for the best configuration and the application of the chosen configuration are tied to successive intervals in *time*. Specifically, to identify the best configuration, the available configurations are typically tested back-to-back one after another. Moreover, once the testing period is over, the adaptation decisions to be made at the beginning of every new interval are based on the behavior of the most recent interval(s). The rationale behind these schemes is that the behavior of the code is largely stable across successive intervals.

In this paper, we introduce a new approach to adaptation: the *Positional* approach. In this case, both the testing for the best configuration and the application of the chosen configuration are associated with *position*, namely with particular code sections. To identify the best configuration, the available configurations are tested on different invocations of the *same* code section. Once the best configuration for a code section is identified, it is kept for later use when that *same* code section is invoked again. The rationale behind this approach is that the behavior of the code is largely stable across invocations of the same section. Note also that, with this approach, we can optimize the adaptations *globally* across the whole program.

To combine ease of implementation and effectiveness, we propose to use *subroutines* as the granularity of code sections in positional adaptation. Moreover, we propose three implementations of subroutine-based positional adaptation that are generic, easy to implement, and effective. Each implementation targets a different workload environment: embedded or specialized server, general purpose, and highly dynamic. Our results show that all three implementations of subroutine-based positional adaptation are much more effective than temporal schemes. On average, they boost the energy savings of applications by 50% and 84% over temporal schemes in two experiments.

This paper is organized as follows: Section 2 describes in more detail subroutine-based positional adaptation; Section 3 presents our three different implementations; Section 4 discusses our evaluation environment; Section 5 evaluates the implementations; Section 6 discusses related work, and Section 7 concludes.

2 Subroutine-Based Positional Adaptation

This paper proposes the *Positional* approach to adaptation, in contrast to the currently-used *Temporal* approach [1, 2, 4, 6, 7, 8, 9, 12, 15, 19, 21]. The fundamental difference between them is their different approach to exploiting program behavior repetition: the temporal approach exploits the similarity between *successive intervals* of code in dynamic order, while the positional approach exploits the similarity between different *invocations* of the same code section.

These two approaches differ on how they test the configurations to identify the best one, and on how they apply the best configuration. Specifically, temporal schemes typically test several configurations in time succession. Consequently, each configuration is tested on a different section of the code, which may have a different behavior. This increases the inaccuracy of the calibration. Moreover, once the testing period is over, the adaptation decisions to be made at the beginning of every new interval are based on the behavior of the most recent interval(s). As a result, if the code behavior changes across intervals, the configuration applied will be non-optimal.

In positional schemes, instead, we associate both the testing of configurations and the application of the chosen one with *position*, that is, with a particular code section. To determine the best configuration for a code section, different configurations are tested on different executions of the same code section. Once the best configuration is determined, it is applied on future executions of the same code section.

Positional adaptation is based upon the intuition that program behavior at a given time is mostly related to the code that it is executing at that time. This intuition is also explored in [23]. Further, positional adaptation has the advantage that, if we can estimate the relative weight of each code section in the program, we can optimize the adaptations *globally* across the program: each configuration is activated in the code sections where it has the greatest benefit compared to all other sections and all other configurations available, all subject to a maximum cost (e.g. slowdown) for the whole program.

2.1 Granularity of Code Sections

Before applying configurations on code sections, we need to determine the granularity of these sections. Code sections used in positional adaptation should satisfy three conditions: capture homogeneous behavior within a section, capture heterogeneous behavior across contiguous sections, and be easy to support.

In this paper, we propose to use the major subroutines of the application as code sections for adaptation. Intuitively, choosing subroutines is likely to satisfy the first two conditions. Indeed, a subroutine often performs a single, logically distinct operation. As a result, it may well exhibit a behavior that is roughly homogeneous and different from the other subroutines. Later in the paper, we show data that suggest that, on average, code behavior is quite homogeneous within a subroutine (Section 5.2), and fairly heterogeneous across subroutines (Section 5.3.2).

Using subroutines also eases the implementation in many ways. First, most subroutine boundaries are trivial to find, as they are marked by call and return instructions. Second, most applications are structured with subroutines.

In our proposal, each code section is constructed with one of the major subroutines of the application plus all the minor subroutines that it dynamically calls. Code sections can nest one another. The remainder of the program, which is the root (*main*) subroutine and the minor subroutines called from *main*, also forms one code section. This section is usually unimportant: on average, it accounts for about 1% of the execution time in our applications. It is possible that, in some applications, this section may have a significant weight. In that case, we can extend our algorithm to subdivide this section to capture behavior variability. For the applications that we study, this is unnecessary.

It is possible that a given subroutine executes two logically distinct operations, or that it executes completely different code in different invocations. Our algorithms do not make any special provision for these cases and still obtain good results (Section 5).

Finally, there are other choices for code sections, such as fixed-sized code chunks (e.g. a page of instructions) or finer-grained entities such as loops. However, they all have some drawbacks. Specifically, for fixed-sized code chunks, the boundaries are arbitrary and do not naturally coincide with behavior changes. On the other hand, using fine-grained or sophisticated entities may involve higher time or energy overheads. Moreover, there is some evidence that using finer-grained entities only provides fairly limited improvements over using subroutine-based sections [11, 18].

3 Implementing Subroutine-Based Positional Adaptation

We present three different implementations of subroutine-based positional adaptation. They differ on how many of the adaptation decisions are made statically and how many are made at run time. Specifically, we call *Instrumentation* (I) the selection of *when* to adapt the processor, and *Decision* (D) the selection of *what* LPTs to activate or deactivate at those times. Then, each selection can be made *Statically* (S) before execution or *Dynamically* (D) at run time. Each of the three implementations targets a different workload environment, which we label as embedded or specialized-server, general purpose, and highly dynamic (Figure 1).

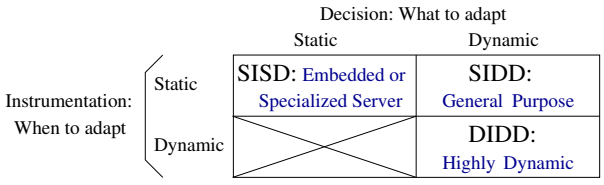


Figure 1. Different implementations of subroutine-based positional adaptation and workload environments targeted.

In general, as we go from *Static Instrumentation and Static Decision* (SISD) to SIDD, and then to DIDD, the adaptation process becomes increasingly automated and has more general applicability. However, it also requires more run-time support and has less global information. Note that there is no DISD environment because the decisions on LPT activation or deactivation cannot be made prior to deciding on the instrumentation points.

We want implementations that are generic, flexible to use, and simple. In particular, they should be able to manage any number of dynamic LPTs. Moreover, to trigger adaptations, we prefer not to use any LPT-specific metrics such as cache miss rate or functional unit utilization. There are two reasons for this. First, it is hard to

cross-compare the impact of two different LPTs using two different metrics. Second, such metrics need empirical thresholds that are often application-dependent.

While positional adaptation can be used for different purposes, here we will use it to minimize the energy consumed in the processor subject to a given tolerable application slowdown (*slack*). We assume that the processor provides support to measure energy consumption. While energy counters do not yet exist in modern processors, it has been shown that energy consumption could be estimated using existing performance counters [16]. In the following, we present each of our three implementations in turn.

3.1 Static Instrumentation & Decision (SISD)

In an embedded or specialized-server environment, we can use off-line profiling to identify the important subroutines in the application, and to decide what LPTs to activate or deactivate at their entry and exit points.

3.1.1 Instrumentation Algorithm

A single off-line profiling run is used to identify the major subroutines in the application. For a subroutine to qualify as major, its contribution to the total execution time has to be at least th_{weight}, and its average execution time per invocation at least th_{grain}. The reason for the latter is that adaptation always incurs overhead, and thus very frequent adaptation should be avoided. We instrument entry and exit points in major subroutines. At run time, minor subroutines will be dynamically included as part of the closest major subroutine up the call graph. Finally, recall that the remaining *main* code in the program also form one "major subroutine".

To reduce overhead, we use several optimizations. For example, we create a wrapper around a recursive subroutine and only instrument the wrapper. Also, if a subroutine is invoked inside a tight loop, we instrument the loop instead.

3.1.2 Decision Algorithm

We perform off-line profiling of the application to determine the impact of the LPTs. Consider first the case where each LPT only has two states (on and off), and LPTs do not interfere with each other. In this case, if the processor supports n LPTs, we perform $n+1$ profiling runs: one run with each LPT activated for the whole execution, and one run with no LPT activated. In each run, we record the execution time and energy consumed by each of the instrumented subroutines. Consider subroutine i and assume that E_i and D_i are the energy consumption and execution time (delay), respectively, of all combined invocations of the subroutine when no LPT is activated. Assume that when LPT_j is activated, the energy consumed and execution time of all invocations of the subroutine is E_{ij} and D_{ij}, respectively. Thus, the impact of LPT_j on subroutine i is ΔE_{ij} and ΔD_{ij}, where $\Delta E_{ij} = E_i - E_{ij}$ and $\Delta D_{ij} = D_{ij} - D_i$. These values are usually positive, since LPTs tend to save energy and slow down execution.

Once we have ΔE_{ij} and ΔD_{ij} for a subroutine-LPT pair, we compute the *Efficiency Score* of the pair as:

$$\begin{cases} -1 & \text{if } \Delta E_{ij} \leq 0 \quad \text{; increases energy consumed} \\ +\infty & \text{if } \Delta E_{ij} > 0 \ \& \ \Delta D_{ij} \leq 0 \quad \text{; saves energy, speeds up} \\ \frac{\Delta E_{ij}}{\Delta D_{ij}} & \textit{Otherwise} \quad \text{; saves energy, slows down} \end{cases}$$

The efficiency score indicates how much energy a pair can save per unit time increase, allowing direct comparisons between different pairs. High, positive values indicate more efficient tradeoffs. Subroutine-LPT pairs that both save energy and speed up the application are very desirable; pairs that increase the energy consumed are undesirable.

Once the results of all subroutine-LPT pairs are obtained, we sort them in a *Score Table* in order of decreasing efficiency score. Each row in the table includes the accumulated slowdown, which is the sum of the ΔD_{ij} of all the pairs up to (and including) this entry. This accumulated slowdown is stored as a percentage of total execution time. This table is then included in the binary of the application, and will be dynamically accessed at run time from the instrumentation points identified above. Note that, in each production run of the application, a tolerable slack for the application will be given. That slack will be compared at run time to the accumulated slowdown column of the table, and a cut-off line will be drawn in the table at the point where the slowdown equals the slack. Pairs in the table that are below the cut-off line are not activated in that run.

Simple extensions handle the case when an LPT has multiple states. Briefly, we perform a profile run for each configuration and record ΔE_{ij} and ΔD_{ij}. Since two such configurations cannot be activated concurrently on the same subroutine, if we select a second configuration, we need to "reverse" the impact of the first configuration on the score table. This effect is achieved by subtracting in the table the impact of one configuration from that of the next most efficient configuration of the same subroutine-LPT pair. In the case where an LPT has too many configurations, the algorithm chooses to profile only a representative subset of them. Alternatively, it could find the best configurations through statistical profiling [6].

When two LPTs interfere with each other or are incompatible in some ways, the algorithm simply combines them into a single LPT that takes multiple states. Some of these states may have ΔE_{ij} and ΔD_{ij} that are not the simple addition of its component LPTs'; other states may be missing due to incompatibility. The resulting multi-state LPT is treated as indicated above.

Finally, we assume that the effect of an LPT on a subroutine is largely independent of what LPTs are activated for *other* subroutines.

3.2 Static Instrum. & Dynamic Decision (SIDD)

In a general-purpose environment, it may be unreasonable to require so many profiling runs. Consequently, in SIDD, only the Instrumentation algorithm is executed off-line. It needs a single profiling run to identify the subroutines to instrument and their weight. The Decision algorithm is performed during execution, using code included in the binary of the application.

The Decision algorithm runs in the first few invocations of the subroutines marked by the Instrumentation algorithm. Consider one such subroutine. To warm up state, we ignore its first invocation in the program. In the second invocation, we record the number of instructions executed, the energy consumed, and the time taken. Then, in each of the n subsequent invocations, we activate one of the n LPTs, and record the same parameters. When a subroutine has gone through all these runs, our algorithm computes the efficiency scores for the subroutine with each of the LPTs, and inserts them in the sorted score table. With this information, and the weight of the

subroutine as given by the Instrumentation algorithm, the system recomputes the new cut-off line in the score table. At any time in the execution of a program, the entries in the score table that are above the cut-off line are used to trigger adaptations in the processor.

The fact that the Decision algorithm runs on-line requires that we change it a bit relative to that in Section 3.1.2. One difficulty is that the efficiency score for a subroutine-LPT pair is now computed based on a single invocation of the subroutine. To be able to compare across invocations of the same subroutine with different numbers of instructions, we normalize energy and execution time to the number of instructions executed in the invocation. Thus, we use Energy Per Instruction (EPI) and Cycles Per Instruction (CPI). Furthermore, the ratio of energy savings to time penalty used in the efficiency score (now $\frac{\Delta EPI_{ij}}{\Delta CPI_{ij}}$) is too sensitive to noise in the denominator that may occur across invocations of the same subroutine. Consequently, we use an efficiency score that is less subject to noise, namely $\frac{EPI_i * CPI_i}{EPI_{ij} * CPI_{ij}}$. The values in the numerator correspond to subroutine i when no LPT is activated, while the values in the denominator correspond to subroutine i when LPT_j is activated. As usual, high efficiency scores are better.

A second difficulty in the on-line Decision algorithm occurs when a subroutine has only a few, long invocations. In this case, the algorithm may take too long to complete for that subroutine. To solve this problem, our system times out when a subroutine has been executing for too long. At that point, our system assumes that a new invocation of the same subroutine is starting and, therefore, it tests a new LPT.

The computation of efficiency scores and the updates to the table only occur in the first few invocations of the subroutines. In steady state, the overhead is the same as in SISD: at instrumentation points, the algorithm simply checks the table to decide what LPTs to activate. Appendix A briefly discusses the overheads involved.

3.3 Dynamic Instrumentation & Decision (DIDD)

We now consider an environment where the application binaries remain *unmodified*. In this case, both Instrumentation and Decision algorithms run on-line. In practice, DIDD is useful in highly-dynamic environments, such as internet workloads where programs are sometimes executed only once, or in just-in-time compilation frameworks. Moreover, it is also useful when it is too costly to modify the binary.

DIDD needs three micro-architectural features. The first one dynamically identifies the important subroutines in the application with low overhead. The second one automatically activates the correct set of LPTs for these subroutines. The third one automatically redirects execution from the first few invocations of these subroutines to a dynamically linked library that implements the Decision algorithm.

3.3.1 Identifying Important Subroutines

We propose a simple micro-architecture module called the *Call Stack* (Figure 2). On a subroutine call, the Call Stack pushes in an entry with the subroutine ID and the current readings of the time and energy counters. For ID, we use the block address of the first instruction of the subroutine. On a subroutine return, the Call Stack pops out an entry. If the difference between the current time and the entry's time is at least th_{invoc}, the subroutine is considered important. As a result, the hardware saves its ID in a fully-associative table of important subroutines called *Call Cache* (Figure 2).

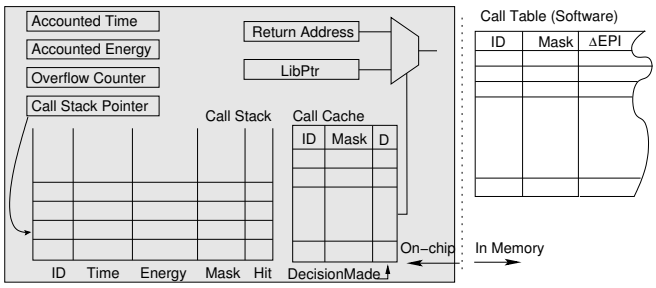


Figure 2. Support for DIDD. The shaded area corresponds to the proposed on-chip hardware. It occupies an insignificant area: about 0.29 mm^2 in 0.18 μm technology.

The Call Stack handles the nesting of important subroutines by subtracting the callee's execution time from the caller's. To support this case, we maintain an *Accounted Time* register that accumulates the cycles consumed by completed invocations of important subroutines up to the current time (Figure 2). With this support, when we push/pop an entry into/from the Call Stack, the current time is taken to be the wall clock time minus the Accounted Time. Moreover, when we finish processing the popping of an important subroutine, we add its time contribution to the Accounted Time. As a result, its contribution will not be erroneously assigned to its caller. Figure 2 also includes an *Accounted Energy* register that is used in the same way.

If the Call Stack overflows, we stop pushing entries. We maintain an *Overflow Counter* to count the number of overflow levels. When the counter falls back to zero, we resume operating the Call Stack.

3.3.2 Activating LPTs & Invoking Decision Algorithm

The hardware must perform two operations at the entry and exit points of these important subroutines. In steady state, it must activate or deactivate LPTs; in the first few invocations of these subroutines, it must redirect execution to the library that implements the Decision algorithm. We consider these two cases in turn.

In steady state, each of the subroutines in the Call Cache keeps a mask with the set of LPTs to activate on invocation (*Mask* field) and a *DecisionMade (D)* bit set. When an entry is pushed into the Call Stack, the hardware checks the Call Cache for a match. If the entry is found, the LPTs in the Mask field are activated (after saving the current mask of LPTs in the Call Stack) and the *Hit* bit in the Call Stack is set. The Hit bit will be useful later. Specifically, when the entry is popped from the Call Stack, if the Hit bit is set, the Call Cache is checked. If the corresponding D bit is set, the hardware simply restores the saved mask of activated LPTs.

In the first few invocations of an important subroutine, the subroutine must run under each of the LPTs in sequence, and the result must be analyzed by the Decision algorithm (Section 3.2). During this period, the corresponding entry in the Call Cache keeps its D bit clear and Mask indicates the single LPT to test in the next invocation. As usual, when an entry is pushed into the Call Stack, the hardware checks the Call Cache and, if the entry is found, the LPT

in the Mask is activated. When the entry is popped out of the Call Stack, if the Hit bit is set and the corresponding D bit is clear, the hardware redirects execution to the Decision algorithm library.

This redirection is done transparently. We modify the branch unit such that when a subroutine return instruction is executed, the hardware checks if the returning subroutine is in the Call Cache and its D bit is clear. If so, the return instruction is replaced by a jump to the entry point of the Decision library code. This entry point is stored in the $LibPtr$ special register (Figure 2). The library runs the Decision algorithm as in Section 3.2: it reads the time and energy consumed by this subroutine-LPT pair from the Call Stack, computes the efficiency score, and updates the score table. In its operation, the Decision algorithm keeps its state in a software data structure in memory called *Call Table* (Figure 2). Once finished, the library issues a return, which redirects execution back to the caller of the important subroutine. This is feasible because the original return address was kept in place, in its register or stack location. In addition, during the redirection to the library, the RAS (Return Address Stack) was prevented from adjusting the pointer and becoming misaligned. With this support, we have effectively delayed the return from the important subroutine by invoking the Decision algorithm seamlessly and with little overhead.

Before the Decision library returns, it updates the Mask for this subroutine in the Call Cache to prepare for the next invocation. If it finds that it has tested all the LPTs for this subroutine, it computes the steady state value for the Mask. Then, it sets the Mask to that value and sets the D bit. Future invocations of this subroutine will not invoke the library anymore.

It is possible that capacity limitations force the displacement of an entry from the Call Cache. There is no need to write back any data. When the corresponding subroutine is invoked again, the Call Cache will miss and, on return from the subroutine, the Decision library will be invoked. At that point, the Decision library will copy the Call Table entry for the subroutine to the Call Cache, effectively restoring it to its old value.

Overall, the proposed hardware is fairly modest. We use CACTI [24] to estimate that the hardware in Figure 2 takes 0.29 mm^2 in 0.18 μm technology. This estimate assumes 32 entries for the Call Cache and Call Stack.

3.3.3 Decision Algorithm

The Decision algorithm used is similar to the one for SIDD. The only difference is that subroutines are now identified on the fly and, therefore, their contribution to the total execution time of the program is unknown. Consequently, the algorithm needs to make a rough estimation. It assumes that all the important subroutines have the same weight: 10%. The inaccuracy of this assumption does not affect the ranking of the subroutine-LPT pairs in the score table. However, it affects the location of the cut-off line in the table. As a result, it is now more challenging to fine tune the total program slowdown to be close to the allowed slack.

We have attempted to use more accurate, yet more costly ways of estimating the contribution of each subroutine. Specifically, we have added support for the system to continuously record and accumulate the execution time of each subroutine. We can then recompute the weights of all the subroutines periodically and update the cut-off line in the score table. From the results of experiments not presented here, we find it hard to justify using these higher-overhead schemes.

3.4 Tradeoffs

Table 1 summarizes the tradeoffs between our three implementations. SISD is the choice when the off-line profiling effort can be amortized over many runs on the platform where profiling occurred. SISD has complete global information of the program and, therefore, can make well-informed adaptation decisions. The only source of inaccuracy is the difference between the profiling and production input sets. Finally, SISD has minimal run-time overhead.

	Pros	Cons	Domain
SISD	Global information of the program. Minimal run-time overhead	Requires many off-line profiling runs. Profiling has to be on target platform	Embedded systems and specialized servers
SIDD	Single performance-only profiling run. Profiling is partially platform independent	Run-time overhead at start-up. Partial information. Limited profiling	General purpose
DIDD	No off-line profiling	Same as SIDD. Extra micro-architectural support	Unavailable off-line profiling: e.g. dynamically generated binary

Table 1. Tradeoffs between the different implementations of subroutine-based positional adaptation.

SIDD has a wider applicability. It is best for environments where software is compiled for a range of adaptive architectures, each of which may even have a different set of LPTs. In this case, the ranking of adaptations is not included in the application code. It is obtained on-line, by measuring the impact of each LPT on the target platform, while the application is running. The only off-line profiling needed is to identify important subroutines and their execution time weight. This does not need to be carried out on the exact target platform. However, SIDD has several shortcomings. First, it incurs run-time overhead, partly due to the application of inefficient adaptations during the initial period of LPT testing. Second, some decisions on what adaptations to apply are necessarily sub-optimal, since they are made before testing all subroutine-LPT pairs. Finally, SIDD relies on the first few invocations of each subroutine to be representative of the steady state, which may not be fully accurate.

DIDD has the widest applicability. It works even when no off-line profiling is available. This is the case when binaries are dynamically generated, or in internet workloads where programs are often executed only once. The shortcomings of DIDD are the micro-architectural support required and all the shortcomings of SIDD with higher intensity. In particular, identifying the important subroutines on-line is challenging and, unless it is done carefully, may lead to high overheads.

4 Evaluation Environment

4.1 Architecture and Algorithm Parameters

To evaluate positional adaptation, we use detailed execution-driven simulations. The baseline machine architecture includes a 6-issue out-of-order processor, two levels of caches, and a Rambus-based main memory (Table 2). The processor can be adapted using three LPTs, which are described in Section 4.3. The simulation models resource contention in the entire system in detail, as well as all the overheads in our adaptation algorithms.

We compare our implementations of positional adaptation to three existing temporal adaptation schemes, which we call

Processor			
Frequency: 1GHz			Branch penalty: 8 cycles (min)
Technology: 0.18μm			Up to 1 taken branch/cycle
Voltage: 1.67V			RAS entries: 32
Fetch/issue width: 6/6			BTB: 2K entries, 4-way assoc
I-window entries: 96			Branch predictor:
Ld/St units: 2			gshare
Int/FP/branch units: 4/4/1			entries: 8K
MSHRs: 24			TLB: like MIPS R10000
Cache	L1	L2	Bus & Memory
Size:	32KB	512KB	FSB frequency: 333MHz
RT:	3 cycles	12 cycles	FSB width: 128bit
Assoc:	2-way	8-way	Memory: 2-channel Rambus
Line size:	32B	64B	DRAM bandwidth: 3.2GB/s
Ports:	2	1	Memory RT: 108ns

Table 2. Baseline architecture modeled. MSHR, RAS, FSB and RT stand for Miss Status Handling Register, Return Address Stack, Front-Side Bus, and Round-Trip time from the processor, respectively. Cycle counts are in processor cycles.

DEETM′, *Rochester*, and *Rochester′*. The parameter values used for all the schemes are shown in Table 3. th_{weight}, th_{grain}, and th_{invoc} are set empirically.

Algorithm	Parameter Values
SISD and SIDD	th_{weight} = 5%; th_{grain} = 1,000 cyc; LPT (de)activation overhead: 2-10 instr
DIDD	th_{invoc} = 256 cyc; Call Stack: 32 entries, 9B/entry, 56 pJ/access; Call Cache: 32 entries, full-assoc, 4B/entry, 66 pJ/access
DEETM′	Microcycle = 1, 10, 100 μs; Macrocycle = 1,000 microcycles
Rochester	Parameter values as in [3, 4], e.g. basic interval = 100 μs
Rochester′	Rochester with the tuning optimization in [6]

Table 3. Parameter values used for the positional and temporal adaptation schemes.

Consider the positional schemes first. Under static instrumentation (SISD and SIDD), we filter out subroutines whose average execution time per invocation is below th_{grain}; under dynamic instrumentation (DIDD), we filter out any invocation that takes less than th_{invoc}. These two thresholds have different values because they have slightly different meanings. The table also shows the values of the main instruction and energy overheads of the schemes; they are discussed in Appendix A. The energy numbers are obtained with the models of Section 4.2.

DEETM′ is an enhanced version of the DEETM Slack algorithm in [12]. In this algorithm, the set of active LPTs is re-assessed at constant-sized time intervals called *macrocycles*. At the beginning of a macrocycle, each different configuration is tested for one *microcycle*. After all configurations have been tested in sequence, the algorithm decides what configuration to keep for the remainder of the macrocycle. This algorithm is more flexible than the one in [12]: the latter assumes a fixed effectiveness rank of LPTs, which limits the set of configurations that it can apply. In [12], a microcycle is 1,000 cycles and a macrocycle is 1,000 microcycles. We examine three different microcycles, namely 1, 10, and 100 μs. We call the schemes DEETM′1, DEETM′10, and DEETM′100, respectively.

Rochester is the scheme in [4]. The algorithm uses a basic interval. Initially, each configuration is tested for one interval. After that, the best configuration is selected and applied. From then on, at the end of each interval, the algorithm compares the number of branches and cache misses in the interval against those in the previous interval. If the difference is within a threshold, the configuration is kept, therefore extending the effective interval. If the difference is over the threshold, the algorithm returns to testing the configurations. The algorithm uses several other thresholds. For our experiments, we start with the parameter values proposed by the authors [3], including a basic interval of 100,000 cycles. We then slightly tune them for better performance.

Rochester′ adds one enhancement proposed in [6] to the Rochester scheme. The enhancement appears when the difference between the branches and misses in one interval and those in the previous one is above the threshold. At that point, Rochester′ does not return to testing the configurations right away. The rationale is that it is best not to test configurations while the program goes through a phase change. The algorithm waits until the difference is below the threshold, which indicates that the change has stabilized. Then, the testing of configurations can proceed.

Overall, we consider temporal schemes with fixed-size intervals (DEETM′) and with variable-sized intervals (Rochester and Rochester′). Note that we do not choose the interval sizes so that all schemes have exactly the same average size, or they match the average size of the intervals in positional schemes. Instead, we use the parameter values as proposed by the authors (although we also slightly tune them to get better performance). With this approach, we hope to be fair and capture good design points for each scheme.

4.2 Energy Consumption Estimation

To estimate energy consumption, we incorporated Wattch [5] into our simulator. We enhanced Wattch in two ways. Recall that Wattch uses a modified version of CACTI [24] to model SRAM arrays. We have refined the modeling of such structures to address several limitations. Specifically, we enhanced the modeling of the sense amplifiers and the bitline swing for writes to make them more accurate. In addition, we always search for the SRAM array configuration that has the lowest energy consumption given the timing constraints.

For the energy consumption in the functional units, we used Spice models of the functional units of a simple superscalar core to derive the average energy consumed for each type of operation. We used results from [20] for more complicated functional units.

We compute the energy consumed in the *whole* machine, including processor, instruction and data caches, bus, and main memory. To model the energy consumed in the memory, we use Intel's white paper [14]. For example, from that paper, one memory channel operating at full bandwidth and its memory controller consume 1.2W.

4.3 Adaptive Low-Power Techniques (LPTs)

We model an adaptive processor with three LPTs that can be dynamically activated and deactivated (Figure 3). These LPTs are: a filter cache [17], a phased cache [10] mode for the L1 data cache, and a slave functional unit cluster that can be disabled. We choose these LPTs because they are well understood and target some major sources of energy consumption in processors. Note that our adaptation algorithms are very general and largely independent of the LPTs used – we simply choose these three LPTs as *examples*.

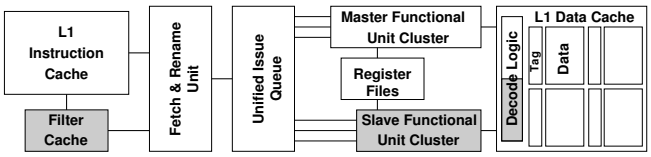


Figure 3. Pipeline of the adaptive processor considered. The shaded areas show the LPTs.

Instruction Filtering (IFilter). The instruction cache hierarchy has a 1-Kbyte filter cache [17]. If it is deactivated, instructions are fetched from the L1 instruction cache; otherwise, the processor checks the filter cache first. Using the filter cache usually saves energy because each hit consumes very little energy. The reason is that the filter cache is small, direct-mapped, and does not require a TLB check because it is virtually tagged. However, the code may run slower because of frequent misses in the small filter cache, which then access L1. Interestingly, this LPT may sometimes speed up the code: when the working set is small enough to fit in the filter cache, the faster access allows quicker recovery on branch mispredictions.

We do not maintain inclusion between the filter and L1 instruction caches. A read access to the filter cache takes 1 cycle and consumes 386 pJ, compared to 2 cycles and 2022 pJ to access the fully-pipelined L1. We model the filling of cold caches.

Phased Cache Mode (PCache). A phased cache is a set-associative cache where an access first activates only the tags [10]. If there is a match, only the matching data array is subsequently activated, reducing the amount of bitline activity and sense amplification in the data array. Consequently, a phased cache saves energy at the cost of extra delay.

In our processor, the 2-way set-associative L1 data cache can work as a normal or as a phased cache. Based on our analysis, in phased mode, a cache hit consumes 974 pJ, a 45% reduction over the 1763 pJ consumed in a normal mode hit. However, the hit takes two extra cycles to complete. Cache misses save even more energy and do not add latency. Note that there is overhead in switching between the two modes. Specifically, when the phased mode is activated, the cache buffers the signal to the data array for two cycles. When the cache is restored to normal mode, the cache blocks for two cycles to drain the pipeline. All these transition overheads are fully modeled.

Reduced Number of ALU Units (RALU). Wide-issue processors typically have many functional units (FUs). Since few applications need all the FUs all the time, processors typically clock-gate unused FUs. Our Wattch-based simulator models the normal clock-gating of unused FUs by reducing the energy consumed by FUs to 10% of their maximum consumption when they are unused.

With the RALU LPT, we go beyond this reduction. In our processor, the FUs are organized into a master and a slave cluster. Each cluster has two FP, two integer, and one load/store unit. The master cluster also has a branch unit. When this LPT is deactivated, both FU clusters can be used, and clock gating proceeds as indicated above. When this LPT is activated, the slave FU cluster is made inaccessible. This allows us to save all the clock distribution energy in the slave cluster. As a result, we save most of the remaining 10% dynamic power in the FUs of the cluster. For multi-cycle FUs, we can only activate this LPT after the FU pipeline is drained. This effect is modeled in our simulations. Overall, this LPT can only have a modest energy-savings effect.

4.4 Applications

To assess positional adaptation on different kinds of workloads, we run multimedia, integer, and floating-point applications. In selecting these applications, we try to include a diverse set of high-level behaviors. In particular, we include programs where the average dynamic subroutine is very short (30 instructions in MCF) or very long (35,000 instructions in HYDRO). The applications are compiled with the IRIX MIPSPro compiler version 7.3 with -O2.

Table 4 lists the applications. Each application has an input set used for the off-line profiling runs (*Profiling*), and one for all other experiments (*Production*). Recall from Section 2 that positional adaptation has the advantage that it optimizes the adaptations *globally*: each configuration is activated in the globally best section of the program. Therefore, to fully demonstrate the effectiveness of positional adaptation, we need to simulate the applications from the beginning to the end. Unfortunately, the full *ref* SPEC input sets are too large for this. Consequently, as the production input sets for the SPEC applications, we use a reduced reference input set (*reduced ref*), which enables us to run the simulations to completion. With these inputs, simulations take from several hundred million to over 2.5 billion cycles. For all applications, we have verified that these reduced input sets running on our simulator produce similar cache and TLB miss rates as the *ref* inputs running natively on a MIPS R12000 processor. We have also verified that the relative weight of each subroutine does not change much. For additional verification, one experiment in Section 5.3 compares executions with *reduced ref* and *ref* input sets.

Suite	Application	Profiling Input	Production Input
SPECint2000	BZIP CRAFTY GZIP MCF PARSER	Test	Reduced ref
SPECfp95	HYDRO APSI	Test	Reduced ref
Multimedia	MP3D	128kbps joint	160kbps joint HQ
	MP3E	24kbps mono	128kbps joint

Table 4. Applications executed.

Due to space limitations, we do not show the breakdown of the energy consumed in the different components of our architecture as we run these applications. However, our results broadly agree with other reports [5]. As expected, energy consumption is widely spread over many components. Therefore, it is unlikely that a single LPT can save most of the energy.

5 Evaluation

To evaluate subroutine-based positional adaptation, we first characterize our algorithms (Section 5.1), then evaluate their impact (Section 5.2), and finally show why the subroutine is a good granularity (Section 5.3).

5.1 Characterization

Table 5 shows the result of running our static and dynamic Instrumentation algorithms. Recall that our algorithms identify the major subroutines in the code and instrument their entry and

exit points. At run time, non-major subroutines are automatically lumped in with their caller major subroutines. Also, the *main* code in the program plus any non-major subroutines that it dynamically calls form one other "major subroutine". For comparison, the table also shows data on all the subroutines in the applications.

Applic	Stat Instrum		Dyn Instrum		All Subroutines in Application		
	N	T (μs)	N	T (μs)	N	Size/Invocation	
						Time (ns)	Instruc
APSI	14	8.8	19	6.1	93	200.1	272.3
BZIP	4	2612.0	5	275.9	54	71.4	108.5
CRAFTY	5	2.6	10	11.6	113	49.6	58.9
GZIP	6	2368.0	11	955.1	69	152.7	202.2
HYDRO	8	2530.0	15	407.6	111	51349.4	34784.1
MCF	3	20.3	6	5.0	50	50.4	28.4
MP3D	5	3.5	9	5.4	65	928.9	1411.5
MP3E	7	35.4	24	8.5	151	178.5	280.3
PARSER	7	28.7	62	976.4	267	37.6	39.2
Average	6.5	845.5	17.9	294.6	108.1	5890.9	4131.7

Table 5. Characterizing the static and dynamic Instrumentation algorithms. In the table, N is the number of major subroutines, while T is the time between instrumentation points.

The data shows that our algorithms identify only a handful of major subroutines to drive LPT activation/deactivation. On average, the number is about 7 and 18 for the static and dynamic algorithms, respectively. This suggests that the structures needed to manage adaptation information are small (e.g. a 32-entry Call Cache in DIDD). Static and dynamic algorithms select a different number of subroutines because they work differently.

The table also shows the average time between instrumentation points, as they are found dynamically at run time. The time ranges from a few μs to thousands of μs. This is the average time between potential adaptations. Within one algorithm, this time varies a lot across applications, which indicates a range of application behavior. On average, these time values are roughly of the same order of magnitude as the intervals in temporal schemes (Table 3). They are long enough to render various overheads negligible (Appendix A).

We now characterize the activation of our LPTs. In a series of experiments, we activate each LPT_j on each subroutine i and record the resulting total energy saved in the program (ΔE_{ij}) and total program slowdown (ΔD_{ij}). With these values, we compute the efficiency score (Section 3.1.2) of each subroutine-LPT pair. We then *rank* the pairs from higher to lower efficiency score and accumulate the total energy and total delay. The result is the *Energy-Delay Tradeoff* curve of the application.

Figure 4-(a) shows such a curve for BZIP. The origin in the figure corresponds to a system with no activated LPT. As we follow the curve, we add the contribution of subroutine-LPT pairs from most to least efficient, accumulating energy reduction (Y-axis), and execution slowdown (X-axis). Finally, the last point of the curve has all the LPTs activated all the time. As an example, in Figure 4-(a), we show the contribution of a subroutine-LPT pair that saves ΔE_{ij} and slows down the program ΔD_{ij}.

The curve can be divided into three main regions. In the *Always apply* region, the curve travels left and up. This region contains subroutine-LPT pairs that both save energy and speed up the program. An example may be a filter cache in a small-footprint subroutine with many mispredicted branches. The filter cache satisfies the average access faster and with less energy than the ordinary cache. Overall, we always enable the pairs in this region.

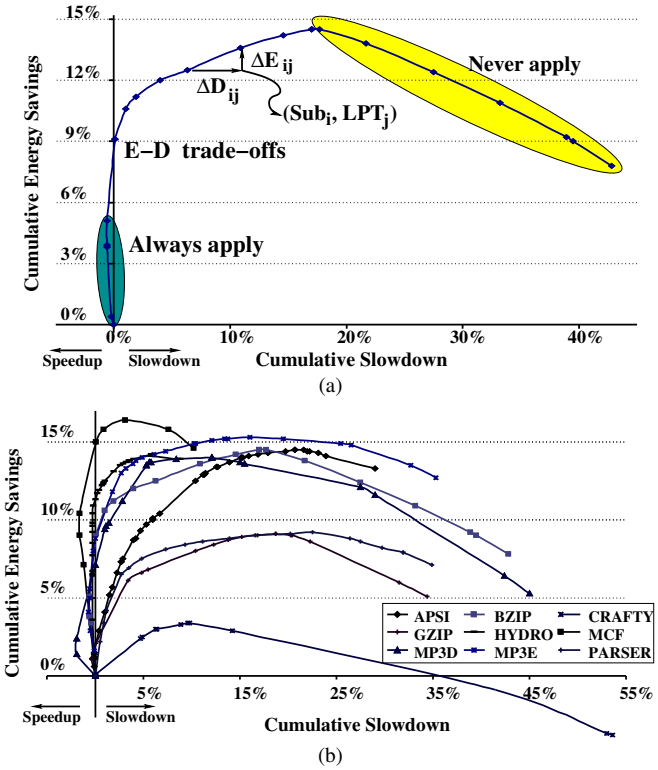


Figure 4. Energy-delay tradeoff curve for BZIP (a) and for all the applications (b).

In the *E-D trade-offs* region, the curve travels right and up. This region contains pairs that save energy at the cost of slowing down the program. This is the most common case. Starting from the left, we apply the pairs in this region until the accumulated application slowdown reaches the allowed slowdown (slack).

In the *Never apply* region, the curve travels right and down. This region contains pairs that increase energy consumption and slow down the program. These pairs should not be applied.

Figure 4-(b) shows the curves for all the applications. We see that all the applications exhibit a similar behavior. The figure also shows that if all LPTs are activated indiscriminately all the time (rightmost point), the result is a very sub-optimal operating point.

Finally, we characterize how our algorithms use the three LPTs. Figure 5 shows the percentage of time that each LPT is activated for the different applications. Due to space constraints, we only show data for SISD. The figure shows the results for a target application slack set to 0.5% and 5% of the application execution time. Overall, the figure shows that our algorithm activates all three LPTs for a significant portion of the time in many applications. Moreover, LPT selection varies across applications.

5.2 Effectiveness of Positional Adaptation

To evaluate the effectiveness of positional adaptation, we perform two experiments, where we want to save as much energy as possible while trying to limit the performance penalty to no more than 0.5% or 5.0%. We compare our three positional schemes (SISD, SIDD, and DIDD) to the temporal algorithms in Table 3 (DEETM′1, DEETM′10, DEETM′100, Rochester, and Rochester′).

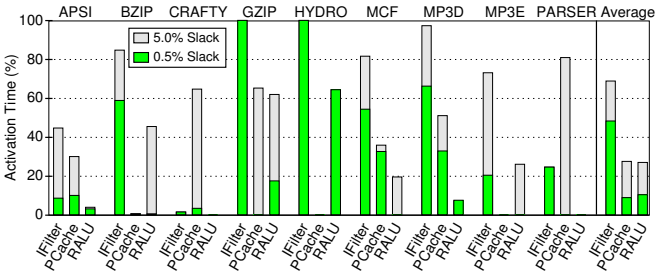


Figure 5. Percentage of time that each LPT is activated in each application under SISD. The data corresponds to two experiments with different slacks.

For each algorithm, Figure 6 shows the reduction in energy in the machine (upper bars) and the increase in execution time of the application (lower bars). The energy includes the contribution of the processor, caches, bus, and memory. The bars are separated into two groups, corresponding to the 0.5% and 5.0% slack experiments. Each bar is the average of our nine applications.

Note that the energy reduction bars are normalized to the energy reduced by an *ideal* adaptation algorithm that serves as an upper bound for a given slack. This algorithm adapts the processor every 1,000 instructions based on *perfect* knowledge of the impact of each of our LPTs on these upcoming instructions. Moreover, the adaptation is overhead-free. We choose to show the bars relative to this ideal scheme rather than to simply show the fraction of energy reduced by each scheme. The reason is that the latter depends on how good *our LPTs* are as much as how good *our algorithms* are. Recall that our algorithms are general and largely independent of the LPTs used. With these LPTs, the ideal algorithm reduces energy use in the machine by 8.7% and 11.6% for the 0.5% and 5.0% slack experiments, respectively. Our bars show how close we get to this ideal reduction.

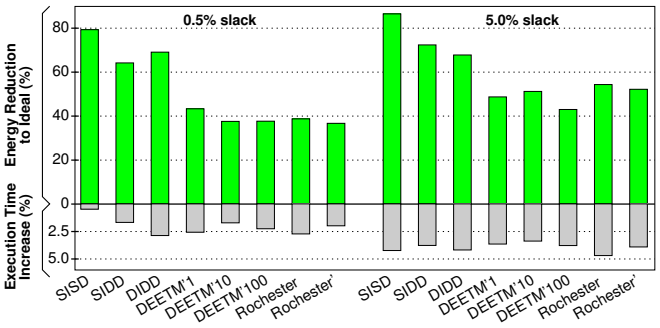


Figure 6. Energy reduction in the machine and program execution time increase for different algorithms.

Consider the 0.5% slack experiments first. In energy savings, the positional schemes are significantly more effective than the temporal ones. On average, positional schemes are 70% as effective as the ideal scheme, while temporal schemes are only 38% as effective. Positional schemes are more effective because they are able to predict code behavior more accurately. As discussed in Section 2, the accuracy is greater for two reasons: (1) in the testing period, they test all configurations on different invocations of the *same* code, and (2) in steady state, they make the adaptation decisions for an upcoming interval based on the behavior of a previous instance of the *same* interval.

Among positional schemes, SISD saves about 80% of the energy that the ideal scheme saves, and does not slow down the program much beyond the target slack. It is, therefore, the preferred scheme if it is possible to use it. Both SIDD and DIDD save less energy and mispredict past the 0.5% slack. In particular, DIDD mispredicts significantly, mostly because of lack of global information at run time. With such a slowdown, DIDD manages higher energy savings than SIDD. In normal conditions, we would expect the opposite.

For the 5% slack case, the positional schemes are again more effective than the temporal ones: on average, they save 75% of the energy that the ideal scheme saves, while temporal schemes save 50%. Among the positional schemes, there is a more gradual change in behavior. The smoother shape appears because positional schemes can now identify good subroutine-LPT pairs to apply more easily than in the 0.5% slack experiment. To see why, note that we want the pairs in Figure 4-(b) that are to the left of *X=5%* (instead of those to the left of *X=0.5%* in the 0.5% slack experiment). The wider range available lessens the impact of measurement inaccuracies, causing fewer selections of pairs beyond the target range, as it happened for DIDD in the 0.5% slack experiment. Overall, the differences in the resulting SISD, SIDD, and DIDD bars now broadly reflect the difference in accuracy between the schemes.

As for the temporal schemes, the differences in energy and slowdown between them appear to be modest. We note that each scheme has its own strengths. Specifically, Rochester and Rochester′ can vary the size of the interval between adaptations dynamically, which improves their effectiveness. On the other hand, the DEETM′ schemes have the ability to apply any given LPT for only a fraction of a macrocycle [12], if they estimate that full application would result in exceeding the slack. The result is that all the schemes have roughly similar effectiveness.

Overall, we derive two main conclusions. First, positional schemes are more effective than temporal ones. They boost the energy savings over temporal schemes by an average of 84% and 50% in the two experiments performed. Moreover, they are relatively more effective in the small slack experiment, where accurately selecting the best adaptation is harder.

The second conclusion results from the observation that the energy savings of the ideal and SISD schemes are quite close (on average, SISD saves 83% of ideal). Recall that the ideal scheme selects the best LPTs every 1,000 dynamic instructions without overhead, while SISD can only attempt to select at major subroutine boundaries. Such boundaries occur every several hundred μs on average (Table 5). Consequently, we infer that the behavior of the code executed inside each of these subroutines appears quite homogeneous to our LPTs.

5.3 Insights into Subroutine-Based Adaptation

Finally, we present data to help understand why subroutine-based positional adaptation is effective. Specifically, we discuss its accuracy in the testing (Section 5.3.1) and steady-state (Section 5.3.2) periods. We also discuss the influence of different input sets (Section 5.3.3). Appendix A discusses its overheads.

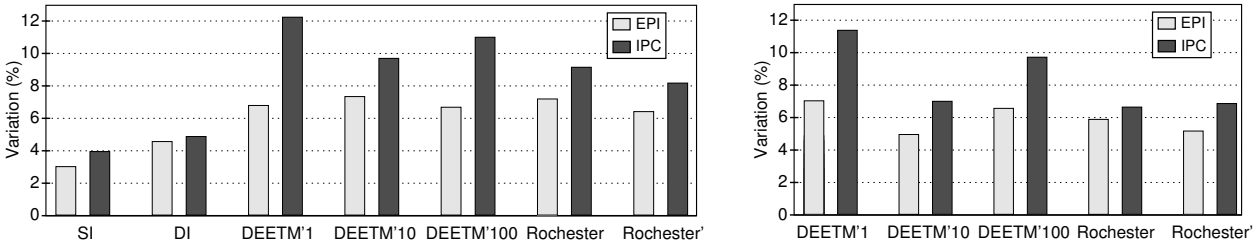


Figure 7. Variation of Energy Per Instruction (EPI) and IPC across testing intervals without actually applying any LPTs. Chart (a) uses the usual input sets for the applications, while Chart (b) changes the input sets of the SPEC applications to be *ref*.

5.3.1 Accuracy in the Testing Period

The accuracy of an adaptation algorithm is affected by how accurately the different configurations are calibrated during the testing period. Recall that, in that period, each configuration is activated for one interval. Then, the impact of the different configurations are compared to each other. Since each configuration is tested on a different interval, the more stable the code behavior is across these intervals, the more accurate this comparison is.

To estimate code stability across testing intervals, we measure the average energy per instruction (EPI) and IPC of each testing interval without actually applying any LPTs. Then, we compute the variation of these metrics in the testing period.

Figure 7 shows the resulting variation of the EPI and IPC for different algorithms. For positional adaptation, we consider the static (*SI*) and dynamic (*DI*) Instrumentation algorithms. For these algorithms, the testing intervals are the first few invocations of each major subroutine. For DEETM$'$, the testing intervals are the first few microcycles in each macrocycle. We compute the average on a macrocycle basis and then average out for all macrocycles. Finally, for Rochester and Rochester$'$, every phase change is followed by several testing intervals. Consequently, we compute the average on a phase-change basis and then average out for all phase changes.

The figure is divided in two parts. Consider first Figure 7-(a), which uses default parameters. We see that the subroutine-based positional algorithms have lower EPI and IPC variations. This suggests that they test configurations during more stable execution conditions and, therefore, achieve a higher accuracy in calibrating LPTs. The reason is that they test all configurations on the *same* code section.

Recall that to fully evaluate our positional schemes, we need to run applications to completion and, therefore, had to reduce the input sets for the SPEC applications. To see if using a bigger input set affects the results, Figure 7-(b) repeats the experiments using the *ref* input sets for the SPEC applications. The programs run for a window of 4.1 billion cycles after the initialization, and then stop. In this case, only the temporal schemes can be evaluated fairly. From the figure, we see that using the bigger input sets reduces the variation in the temporal schemes only slightly.

5.3.2 Accuracy in the Steady-State Period

Adaptation algorithms identify steady-state periods where a configuration is kept unchanged from one interval to the next. Code conditions are stable, and the algorithm predicts that the current configuration will have a similar impact in the next interval. Consequently, the steady-state accuracy of an algorithm will be greatest when the impacts of a configuration on two intervals that belong to the same steady state are most similar.

To estimate the accuracy, we identify, for each algorithm, the set of intervals that it considers to be in a given steady state. For positional algorithms, these are successive invocations of the same subroutine in steady state; for temporal algorithms, they are contiguous intervals not separated by phase changes. We then apply one LPT to all these intervals and record the changes in energy consumption and execution speed. Then, we compute the variation of this change across all these intervals. The smaller this variation is, the more accurate the scheme is in steady state. Finally, we average out all the steady states.

Figure 8 shows the variation for different algorithms. For brevity, we only show the average of all three LPTs. From the figure, we see that the subroutine-based positional algorithms have a lower variation. The impact of LPTs across intervals in steady state is more stable. Consequently, these algorithms can more accurately predict the impact of an LPT on an upcoming interval in steady state. This low variability is a result of executing the same code section.

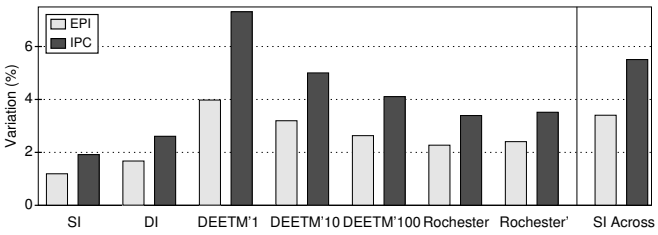


Figure 8. Variation of the impact induced by an LPT on steady-state intervals. The "SI Across" bars on the right show, for SI, the variation of the impact across different subroutines. In all cases, the data is the average for IFilter, PCache, and RALU.

For comparison, the "SI Across" bars on the right of Figure 8 show, for SI, the variation of the impact across different subroutines. We can see that, for SI, the variation across different subroutines is much higher than that across different invocations of the same subroutine. This data shows that code behavior across subroutines is relatively heterogeneous.

5.3.3 Influence of Different Input Sets

To gain insights into the influence of using different input sets for profiling and production runs in SI, we perform the following experiment. We run an application and measure the average impact of a given LPT on a given subroutine. We perform this experiment for four different input sets: *test*, *train*, *ref*, and *reduced ref*. Figure 9 shows the variation observed across these four runs. In the figure, the data is grouped by application, averaging out all the subroutines and all the LPTs.

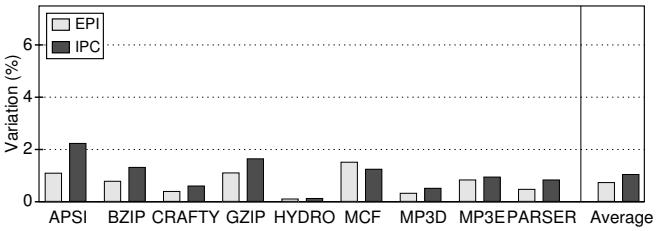


Figure 9. Variation of the average impact of an LPT on a subroutine across different input sets.

Overall, the average impact of an LPT on a subroutine is quite stable across different input sets. In fact, if we compare the figure to the SI bars in Figure 8, we see that the variation across different input sets is even smaller than the variation across invocations of the subroutine inside a single program execution. This suggests that, to predict the average impact of an LPT on a subroutine, it can be more accurate to use the *average* impact measured with a *training* input than to use the impact measured on *one* invocation using the *same* input. Therefore, using profiles is a viable solution for subroutine-based positional adaptation.

6 Related Work

There are many proposals for adaptive hardware mechanisms targeted at performance or energy optimization [1, 2, 4, 6, 7, 8, 9, 12, 13, 15, 19, 21]. They adapt a variety of aspects of the processor, including cache organization, issue width, issue window size, or voltage and frequency. The majority of these schemes use *Temporal* adaptation [1, 2, 4, 7, 8, 9, 12, 19, 21]: the testing of the LPTs and the activation of the chosen LPTs are related to time. In our paper, we introduced *Positional* adaptation, where both the testing and the application are tied to a code section. Our approach exploits the fact that the behavior of the program at a given time is directly related to the code it is executing. This idea is also exploited in [23].

The scheme in [13, 22] performs a form of positional adaptation for multimedia applications composed of repeating frames. The scheme uses the best adaptation for the past frames to predict the adaptation to perform in the next frame. The adaptation is positional because each frame simply uses a different data set to execute the same code. Overall, this work is different than ours in that it requires user knowledge of the frame-based structure of the code. Furthermore, it is specialized for the multimedia domain. Our work applies to general-purpose code and exploits its subroutine structure.

The scheme in [6] is based on a temporal scheme with variable-sized intervals [4]. Calibrations and applications of adaptations are performed in intervals tied to time (number of instructions). However, the scheme collects working-set signatures for the code executed in each interval and saves the configuration used. When the algorithm sees a signature similar to one seen before, it applies the saved configuration. This is done to eliminate the overhead of another testing period. Reusing adaptations when the code may be similar gives the algorithm an interesting positional aspect.

The temporal scheme in [15] also has an aspect of positional adaptation: reconfiguration is only attempted in some known sections of the code. In these sections, the system tests several adaptations *in time sequence* to identify the best configuration — making the scheme intrinsically temporal.

The temporal scheme in [7] improves the accuracy of the testing period for temporal schemes by using "mimic" counters that can predict the effect of multiple configurations without trying them out one by one. However, like other temporal schemes, the algorithm exploits the similarity of behavior across consecutive time intervals. Moreover, it is not clear that this approach of using counters can be exploited for all LPTs.

Performing adaptations at subroutine boundaries was considered in [4]. However, after comparing it to performing adaptations at periodic intervals, the latter was selected, largely due to simplicity.

7 Conclusions and Future Work

This paper has presented Positional adaptation, a new approach where both the testing of configurations and the application of the chosen configurations are associated with particular code sections. We use subroutines as the granularity for such code sections. We have also designed three very general implementations of subroutine-based positional adaptation, which correspond to different choices in the tradeoff between general applicability and effectiveness. To evaluate these implementations, we selected several example dynamic LPTs. Overall, all three implementations are more effective than temporal adaptation schemes. On average, they boost the energy savings of applications by 50% and 84% over temporal schemes in two experiments. In general, of course, the absolute energy savings are highly dependent on the LPTs used. Intuitively, subroutine-based positional adaptation is effective because the system becomes highly predictable: different invocations of the same subroutine usually have similar code behavior, and react similarly to the same adaptation.

While we have used positional adaptation in a low-power environment, we can also apply it to performance-centric designs. In this case, the designer's toolkit could include a set of High Performance Techniques (HPTs) instead of LPTs. The goal would be to adapt the processor by activating each HPT at the best point in the program, such that the program runs as fast as possible without increasing energy consumption beyond a certain limit. Other environments can also use positional adaptation.

Our finding that the behavior of different invocations of the same subroutine is very predictable can be exploited in many ways. Specifically, it can be used to reduce the overhead of costly operations by intelligently applying them to only one of several executions that we expect to behave similarly. These costly operations can be dynamic optimization, cycle-by-cycle architectural simulation, or evaluation of various time-consuming optimizations.

References

[1] D. Albonesi. Selective Cache Ways: On-Demand Cache Resource Allocation. In *International Symposium on Microarchitecture*, pages 248–259, November 1999.

[2] R. Bahar and S. Manne. Power and Energy Reduction Via Pipeline Balancing. In *International Symposium on Computer Architecture*, pages 218–229, May 2001.

[3] R. Balasubramonian. Personal communication. October 2002.

[4] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory Hierarchy Reconfiguration for Energy and Performance in General-Purpose Processor Architectures. In *International Symposium on Microarchitecture*, pages 245–257, December 2000.

[5] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. In *International Symposium on Computer Architecture*, pages 83–94, June 2000.

[6] A. Dhodapkar and J. Smith. Managing Multi-Configuration Hardware via Dynamic Working Set Analysis. In *International Symposium on Computer Architecture*, pages 233–244, May 2002.

[7] S. Dropsho, A. Buyuktosunoglu, R. Balasubramonian, D. Albonesi, S. Dwarkadas, G. Semeraro, G. Magklis, and M. Scott. Integrating Adaptive On-Chip Storage Structures for Reduced Dynamic Power. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 141–152, September 2002.

[8] D. Folegnani and A. González. Energy-Effective Issue Logic. In *International Symposium on Computer Architecture*, pages 230–239, May 2001.

[9] T. Halfhill. Transmeta Breaks x86 Low-Power Barrier. *Microprocessor Report*, 14(2):1,9–18, February 2000.

[10] A. Hasegawa, I. Kawasaki, K. Yamada, S. Yoshioka, S. Kawasaki, and P. Biswas. SH3: High Code Density, Low Power. *IEEE Micro*, 15(6):11–19, December 1995.

[11] M. Huang, J. Renau, and J. Torrellas. Profile Based Energy Reduction for High-Performance Processors. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization*, December 2001.

[12] M. Huang, J. Renau, S. Yoo, and J. Torrellas. A Framework for Dynamic Energy Efficiency and Temperature Management. In *International Symposium on Microarchitecture*, December 2000.

[13] C. Hughes, J. Srinivasan, and S. Adve. Saving Energy with Architectural and Frequency Adaptations for Multimedia Applications. In *International Symposium on Microarchitecture*, pages 250–261, December 2001.

[14] Intel Corporation. *Mobile Power Guidelines 2000, Rev 1.0*, 1998.

[15] A. Iyer and D. Marculescu. Power Aware Microarchitecture Resource Scaling. In *Design, Automation and Test in Europe*, pages 190–196, March 2001.

[16] R. Joseph and M. Martonosi. Run-Time Power Estimation in High Performance Microprocessors. In *International Symposium on Low Power Electronics and Design*, August 2001.

[17] J. Kin, M. Gupta, and W. Mangione-Smith. The Filter Cache: An Energy Efficient Memory Structure. *International Symposium on Microarchitecture*, pages 184–193, December 1997.

[18] G. Magklis, M. Scott, G. Semeraro, D. Albonesi, and S. Dropsho. Profile-based Dynamic Voltage and Frequency Scaling for a Multiple Clock Domain Processor. In *International Symposium on Computer Architecture*, June 2003.

[19] S. Manne, A. Klauser, and D. Grunwald. Pipeline Gating: Speculation Control for Energy Reduction. In *International Symposium on Computer Architecture*, pages 132–141, July 1998.

[20] A. Nannarelli. *Low Power Division and Square Root*. PhD thesis, University of California, Irvine, Department of Electrical and Computer Engineering, June 1999.

[21] D. Ponomarev, G. Kucuk, and K Ghose. Reducing Power Requirements of Instruction Scheduling Through Dynamic Allocation of Multiple Datapath Resources. In *International Symposium on Microarchitecture*, pages 90–101, December 2001.

[22] R. Sasanka, C. Hughes, and S. Adve. Joint Local and Global Hardware Adaptations for Energy. In *International Conference on Architectural Support for Programming Language and Operating Systems*, pages 144–155, October 2002.

[23] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 45–57, October 2002.

[24] S. Wilton and N. Jouppi. CACTI: An Enhanced Cache Access and Cycle Time Model. *IEEE Journal on Solid-State Circuits*, 31(5):677–688, May 1996.

Appendix A: Positional Adaptation Overheads

Our algorithms have both steady-state and initialization overheads. The former occur when activating/deactivating LPTs in steady state, and have three sources.

The first source is extra instructions to activate/deactivate LPTs in SISD and SIDD. Upon entering an instrumented subroutine, we save the current LPT mask and indirectly load the mask to apply from a table. Upon exiting the subroutine, we restore the mask and check that we are not in initialization mode. Thus, depending on the implementation, entering or exiting an instrumented subroutine adds 2-10 instructions. The frequency of such an operation is shown in Table 5. In the worst-case application (CRAFTY), it occurs once every 2.6 μs, although on average it occurs once every 845 μs or 295 μs depending on the Instrumentation algorithm used. Consequently, the resulting energy and time overhead is negligible.

The second source of overhead is the accesses to the Call Stack and Call Cache in DIDD. Upon any subroutine entry, the Call Stack is updated and the Call Cache is read. If this is a major subroutine, its LPT mask is activated and, on subroutine return, the old mask is restored. On subroutine return, the Call Stack is updated. In all these operations, the energy consumed is modeled. For example, Table 3 shows that a Call Stack and a Cache Cache access consume 56 pJ and 66 pJ, respectively. The energy consumed is included in our simulations, and can be shown to be insignificant. Since these structures consume very little energy, even in an environment where subroutine calls are very frequent, this overhead will not become significant.

The third overhead of LPT activation/deactivation is some architectural state transitions that depend on the particular LPT. These overheads are discussed in Section 4.3, and include buffering the signal to the data array (or blocking the cache) for 2 cycles in PCache and filter cache misses in IFilter. The impact of these overheads on performance and energy consumption is included in our simulations.

Finally, the dynamic execution of the Instrumentation algorithm in DIDD and the Decision one in SIDD and DIDD are initialization overheads. Such overheads are in practice negligible because they occur only during the beginning stages of the program. For example, the Decision algorithm invokes a library $n+1$ times for each instrumented subroutine, where n is the number of LPTs. Although many such invocations involve little more than reading and saving the energy and performance counters, we estimate that the average invocation takes 100 instructions. Given that there are on average 7 or 18 instrumented subroutines per application (Table 5), the total cost of the algorithm is less than 10,000 instructions. This is minuscule compared to the program execution time. An analysis of the Instrumentation algorithm shows that its overhead is also minuscule.

A Framework for Dynamic Energy Efficiency and Temperature Management*

Michael Huang†, **Jose Renau**†, **Seung-Moon Yoo**‡, and **Josep Torrellas**†
Department of Computer Science†
Department of Electrical and Computer Engineering‡
University of Illinois at Urbana-Champaign
http://iacoma.cs.uiuc.edu

ABSTRACT

While technology is delivering increasingly sophisticated and powerful chip designs, it is also imposing alarmingly high energy requirements on the chips. One way to address this problem is to manage the energy dynamically. Unfortunately, current dynamic schemes for energy management are relatively limited. In addition, they manage energy either for energy efficiency or for temperature control, but not for both simultaneously.

In this paper, we design and evaluate for the first time an energy-management framework that tackles both energy efficiency and temperature control in a unified manner. We call this general approach Dynamic Energy Efficiency and Temperature Management (DEETM). Our framework combines many energy-management techniques and can activate them individually or in groups in a fine-grained manner according to a given policy. The goal of the framework is two-fold: maximize energy savings without extending application execution time beyond a given tolerable limit, and guarantee that the temperature remains below a given limit while minimizing any resulting slowdown. The framework successfully meets these goals. For example, it delivers a 40% energy reduction with only a 10% application slowdown.

1 INTRODUCTION

Continuous technical advances are fueling the trend toward more sophisticated and powerful chip designs. Such designs, including high-end microprocessors, chip multiprocessors, systems on a chip, and other advanced embedded systems are quickly increasing their functionality and clock rates. Unfortunately, they are also increasing their energy consumption requirements alarmingly.

One way to address this problem is to manage the energy consumed in the chips. There are two main aims of energy management: to ensure that the energy is used efficiently and to guarantee that power consumption is never so high that the chip reaches dangerous temperature levels.

Efficient energy use is desirable in all systems. However, it is critical in portable devices, where battery energy is limited. It is also an important way to reduce cost in systems that have periods of idle time, also called slack [25]. Slack appears not only in interactive and real-time systems; it also occurs in general-purpose environments like web servers or routers with high-end processors where the performance is often bottlenecked by the network.

Likewise, curbing high power consumption to limit high temperatures is useful in all systems. It enables lower-cost packaging and cooling systems for the chips. It also makes the chip more reliable. Finally, it may enable a more aggressive design or a higher clock speed.

To address these two issues, namely energy efficiency and temperature control, many low-power architectural techniques have been proposed and implemented. For example, they include putting the system in sleep mode [28]; scaling the voltage and/or frequency [11, 13, 25]; switching contexts to a job that consumes less power [27]; reconfiguring hardware structures [1]; gating pipeline signals, for example to control speculation [5, 23]; throttling the instruction cache [28]; clock optimizations, including multiple clocks and clock gating [10]; better signal encoding [10]; low power memory design techniques [15] like bank partitioning or divided word line; low power cache design techniques like cache block buffering [33], sub-banking [9, 30], or filter caches [20]; and TLB optimizations [17].

While most of these techniques are likely to be useful for the upcoming, energy-consuming chips, we feel that their effectiveness can be enhanced. To start with, while some of these techniques have been used adaptively [1, 8, 5, 23, 25, 27], many others have been designed to be always active. In reality, for many of the latter, it would be advantageous to turn them on and off dynamically, based on the requirements of the application and the environmental conditions. They could enable useful energy-performance tradeoffs.

In addition, most of these techniques were proposed to work independently of each other. If we combined many of them in a single framework that can activate and deactivate them individually or in groups according to a given policy, the resulting system could be both more powerful and more flexible.

Finally, proposed dynamic approaches have targeted either energy efficiency [1, 8, 23, 25] or temperature control [5, 27] but not both simultaneously. If a multi-technique framework can combine support for both aspects, it can become a fairly complete approach to energy management.

The general approach of dynamically managing energy for both energy efficiency and temperature control we call *Dynamic Energy Efficiency and Temperature Management (DEETM)*. The contribution of this paper is the design and evaluation for the first time of one such DEETM framework. Our framework supports a combination of energy-management techniques. It is implemented with a combination of software and hardware for fine-grained energy management. The framework has two goals: (i) maximize the

*This work was supported in part by the National Science Foundation under grants NSF Young Investigator Award MIP-9457436, MIP-9619351, and CCR-9970488, DARPA Contract DABT63-95-C-0097, and gifts from IBM and Intel.

savings of energy in the chip without extending the execution time of the application beyond a given tolerable limit, and (ii) guarantee that the temperature of the chip remains below a given limit while minimizing any resulting slowdown. In our evaluation, we show that the framework satisfies these goals. For example, it delivers a 40% energy reduction with only a 10% application slowdown.

This paper is organized as follows: Section 2 presents the design and implementation of our framework for DEETM; Section 3 discusses how we evaluate it; Section 4 evaluates the framework; and Section 5 presents related work.

2 A FRAMEWORK FOR DEETM

In this section, we describe our framework for DEETM: its main ideas (Section 2.1), the algorithm used (Section 2.2), the software interface (Section 2.3), some related issues (Section 2.4), and the techniques included in the framework (Section 2.5).

2.1 Main Ideas

Advanced chips can benefit from a dynamic framework that manages energy in a fine-grained manner to accomplish two goals. The first one is *temperature control*: guaranteeing that the temperature of the chip remains below a given limit while minimizing any slowdown. The second goal is *energy efficiency control*: maximizing the savings of energy in the chip without extending the execution of the application beyond a tolerable limit.

For the framework to be versatile, it should include multiple techniques for energy management. Different techniques may target the energy consumption in different components of the chip, for example processor cores, I-caches, D-caches, or DRAM arrays. They may, instead, target the same component but do so with a different energy-performance tradeoff. In such an environment, the framework can dynamically activate the techniques individually, concurrently, gradually with a priority order, or even in a mutually exclusive manner.

As initial support for the framework, we assume that the chip contains a distributed thermal sensor along the lines of the PowerPC [28] and a counter with the number of instructions executed. In addition, it contains two registers, *MaxTemp* and *MaxSlowdn*, which are set in software with the maximum temperature allowed and the maximum job slowdown that can be tolerated, respectively.

2.2 Algorithm Description

Our framework includes two algorithms: a temperature-limiting one called *Thermal* and an energy-saving one called *Slack*. They try to satisfy the first and second goals discussed above, respectively. These algorithms control the activation of a set of energy-management techniques.

At any given time, the set of techniques that are active is called the *Current Set*. These techniques may have been selected by the Thermal or by the Slack algorithm. The set of techniques that are selected by the Thermal algorithm is called the *Thermal Set*.

The two algorithms work as follows. When the Thermal algorithm runs, it compares the current temperature to the temperature limit. Depending on the result, it may add or subtract one technique to or from the Thermal Set. When the Slack algorithm runs, it first deactivates the Current Set to measure the baseline IPC value of the application. Then, it activates the Thermal Set and possibly additional techniques until the new IPC shows that the tolerable slack is used up.

To adapt to changing conditions, these algorithms run periodically. The period between runs we call *Macrocycle*. Since the two algorithms do not need to have the same period, we define a thermal macrocycle and a slack macrocycle (Figure 1-(a)).

The thermal macrocycle should be set roughly to the time taken by the thermal sensor to detect a change in temperature after a technique is activated. Since heat transfer occurs at the ms level [31], the thermal macrocycle has to be of the order of a few ms, possibly 1-15 ms. If the macrocycle is too short, the Thermal algorithm will overreact, since there is not enough time to feel the effect of any newly activated technique. However, if it is too long, we risk damaging the chip with a temperature that is over the limit for too long. The appropriate length of the macrocycle is different in each system. It depends on the heat dissipation characteristics of the chip and the sophistication of the distributed thermal sensor.

Selecting the slack macrocycle is not as delicate. However, since the Slack algorithm decides what fraction of the time to activate each technique for, based solely on the IPC at the beginning of the macrocycle, we need to pay attention to two issues. First, the macrocycle should be short enough not to miss significant changes in application behavior. Otherwise, the resulting slowdown may be very different than initially expected. In practice, a macrocycle of the order of a few ms, possibly 1-15 ms, is appropriate.

The second issue is that slack macrocycles should all have the same duration and not be cut off short. The reason is that, when the Slack algorithm runs, its calculations use the expected duration of the macrocycle to decide the length of time to activate each technique for. Cutting the macrocycle short makes such calculations inaccurate. We will see later how we address this issue.

In the following, we describe the two algorithms in detail. Note that both algorithms want to deliver large energy reductions without excessive slowdowns. Consequently, they prefer techniques that minimize the product of the energy consumed by the application times the execution time (*energy-delay product* [10]). As a result, both algorithms pick the techniques to activate in the same order. Such order follows a ranking set up by the OS or application based on the expected energy-delay product impact of each technique.

Thermal Algorithm

The Thermal algorithm is typically implemented as an interrupt handler in the OS. Alternatively, it could be implemented in hardware. The algorithm is shown in Figure 1-(b). If the thermal sensor indicates a temperature higher than *MaxTemp*, the next highest-priority technique not yet in the Thermal Set is added to it. Otherwise, if it indicates a temperature lower than a low-threshold value *MinTemp*, the lowest-priority technique in the Thermal Set is removed.

If we have added a new technique to the Thermal Set, before leaving the algorithm, we set the Current Set to the maximum of Current and Thermal Sets. This is done to ensure that the new technique is immediately active. If a technique was removed from the Thermal Set, however, it cannot be removed from the Current Set until the Slack algorithm runs.

MinTemp is set to minimize instability. A sophisticated design can keep a different *MinTemp* for each of the techniques. To choose the appropriate *MinTemp* for a given technique, we can use past profiles to estimate the temperature reduction that the technique delivers under usual conditions. Then, we set *MinTemp* to slightly less than *MaxTemp* minus the average value of such a temperature reduction. With this approach,

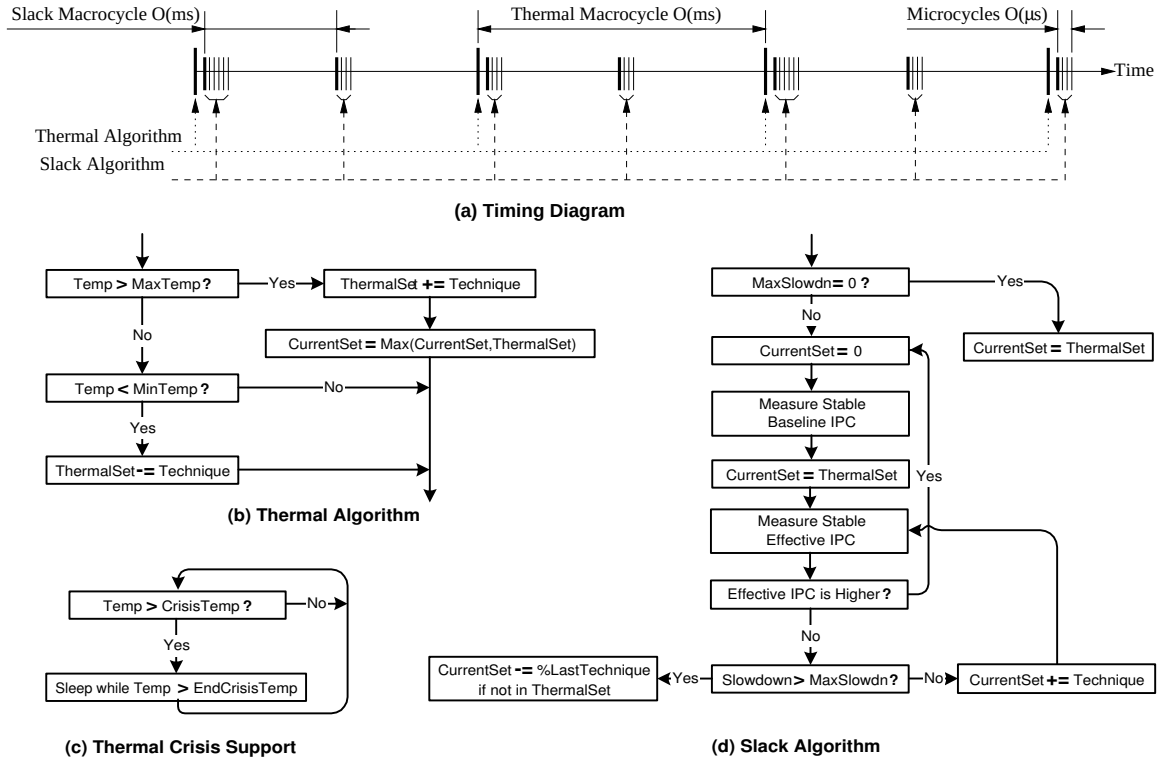


Figure 1: Algorithms used in our framework.

we minimize the chances that the deactivation of a technique brings us back to over *MaxTemp*.

Note that, in some cases, we may not be able to prevent the temperature from rising over the limit. For example, such a situation may be caused by a virus. For this reason, the chip must include support for a thermal crisis. One possible such support is shown in Figure 1-(c): if the temperature reaches a *CrisisTemp* temperature, the hardware unconditionally sets the system to sleep until the temperature is safely lower than *CrisisTemp*.

Slack Algorithm

The Slack algorithm is implemented in hardware instead of as an OS routine. The reason is that, every time that it runs, it needs to repeatedly measure the number of instructions executed by the application at μs-level intervals. After several such measurements in the background, the algorithm makes the decision. These intervals we call *Microcycles* (Figure 1-(a)). We will see that, for higher accuracy, a microcycle is of the order of a few μs.

The Slack algorithm is shown in Figure 1-(d). If no slowdown can be tolerated, the Current Set is simply set to the Thermal Set. Otherwise, the Current Set is deactivated so that the hardware can measure the stable baseline IPC of the application. To compute the IPC, the hardware reads at microcycle intervals the counter of instructions executed. It may take several readings until a reasonably stable IPC is obtained. Note that by deactivating all techniques for several μs we do not risk a dangerous temperature surge because the time is too short.

We then set the Current Set to the Thermal Set and, to find out the resulting slowdown, calculate the new *effective* IPC. The new effective IPC is the new measured IPC plus a correction if the Thermal Set includes techniques that change the clock frequency.

With this new effective IPC, we can compare the slowdown caused by setting the Current Set to the Thermal one, to the maximum tolerable slowdown (*MaxSlowdn*). If *MaxSlowdn* is higher, we augment the Current Set with the next highest-priority technique not yet in it and again measure the effective IPC. This process is repeated until the application slowdown is equal to or higher than *MaxSlowdn*. If the slowdown is higher than *MaxSlowdn*, the last technique that has been added to the Current Set is marked as active for only a fraction of the Slack macrocycle, such that the final slowdown ends up being no higher than *MaxSlowdn*. The only exception is when this last technique added belongs to the Thermal Set, in which case, it cannot be deactivated. Finally, when we reach this point, the algorithm exits.

Every time that we go through the loop of adding a new technique to the Current Set, the hardware may need to take several measurements spaced one microcycle apart, until a stable IPC is obtained. Unfortunately, it is possible that, at the same time, the application also goes through a change in its regime that induces a change in IPC. In this case, to avoid confusing our algorithm, we proceed as follows. If the effective IPC suddenly becomes higher after activating a technique, it is clear that the regime changed. If we pressed on with more techniques until we reached the original target IPC, we would be slowing down the application beyond the tolerable limit. Consequently, as shown in Figure 1-(d), we stop the algorithm and restart it from the beginning.

If, instead, the regime change is in the opposite direction, our algorithm will not notice it: we will assume that the technique just activated is solely responsible for the large IPC reduction. However, this is fine. Our algorithm will end up producing a conservative solution: in the final system, the true slowdown relative to the baseline execution will be less than it could be tolerated. Consequently, the end user is not negatively affected.

Note that some of the techniques used may have non-trivial activation delays. Such is the case, for example, for voltage-frequency scaling, which takes 10-20 μs to activate or deactivate [11]. Such delays, however, are negligible compared to the duration of a macrocycle. For example, if a slack macrocycle takes 2 ms, activating and deactivating voltage-frequency scaling takes only about 2% of the macrocycle. Furthermore, because the impact of voltage-frequency scaling on the IPC is fairly predictable, we do not need to deactivate it at every beginning of a macrocycle to estimate the baseline IPC. This fact further reduces overhead.

Finally, since both the Thermal and the Slack algorithms may update the Current Set, we need to prevent inconsistencies. To this end, and also to ensure that slack macrocycles are not cut off short, we propose the following timing (Figure 1-(a)). We choose the slack macrocycle so that a thermal one contains several slack macrocycles plus a few μs. After the OS has executed the Thermal algorithm and is about to return execution to user mode, it sets the hardware to trigger the next run of the Slack algorithm in a few μs. We set this delay so that, when the Slack algorithm finally runs, it finds the user application in a warmed-up state. From then on, the Slack algorithm runs periodically, always in the background. Finally, when an interrupt triggers the Thermal algorithm again, the first action of the OS is to temporarily disable the hardware that triggers the Slack algorithm. If it so happens that the Slack algorithm was running at the time, this action stops it and automatically sets the Current Set to the Thermal Set.

2.3 Software Interface

The *MaxTemp* and *MaxSlowdn* registers presented above are part of our framework's software interface. In addition, for each energy-management technique, the interface contains a register with the relative priority of activation of the technique (Figure 2). All registers are set by the OS, although *MaxSlowdn* can also be set by the application. With this support, our algorithms can decide what techniques to include at any time in the Current Set.

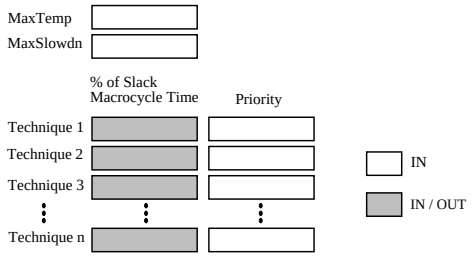


Figure 2: Software interface of our framework.

However, the OS should also have a means to directly overwrite the decisions taken by our default algorithms. This capability can be useful when the OS has specific information on the performance or energy characteristics of the application that is running. Such information may be available from a profile of the application.

One way to extend the interface is to allow the OS to overwrite the decisions of the algorithms as shown in Figure 2. We add one input/output register for each technique in the framework. For a given technique, the register indicates the fraction of the slack macrocycle for which the technique is activated. While these registers are automatically set by the Slack algorithm as it adds techniques to the Current Set, they can also be overwritten by the OS.

2.4 Related Issues

Two important related issues are whether to implement the algorithms in hardware or in software, and whether to make the decisions in a centralized or distributed manner in the chip. We consider these issues next.

2.4.1 Hardware vs Software Implementation

The Thermal algorithm is implemented as an OS interrupt handler. While the Slack algorithm could also be implemented in software, we choose to implement it in hardware. This is in contrast to related algorithms proposed in the literature that exploit system idleness in software [4, 25].

A software implementation of the Slack algorithm would certainly be sufficient if we restricted our work to a certain class of energy-management techniques or to a certain class of applications. Specifically, suppose that we restricted our techniques to those that induce predictable slowdowns like voltage-frequency scaling. In this case, the OS can simply activate the technique for the time duration that will induce the desired slowdown.

Likewise, software might be enough if we restricted the applications to those that, by repeating certain high-level operations, easily tell how fast they are executing. For example, consider video streaming applications. Their speed can be easily monitored by recording the number of frames per unit of time that are being processed. It is easy for the OS to know what is the slowdown caused by a certain energy-management technique by simply checking the new frame rate. There is no need to measure the IPC.

However, we want our Slack algorithm to deliver accurate solutions for all classes of techniques and applications. To see why it requires a hardware implementation, recall that the Slack algorithm repeatedly measures the IPC of the application. While software can support measurements at ms-level intervals, only a hardware solution can support measurements at μs-level intervals. In practice, we need a hardware solution only if the behavior of the application changes significantly at ms-level intervals while staying relatively uniform at μs-level intervals.

We have evidence that μs-level measurements are beneficial in our applications. To understand why, consider a loop. In general, IPC measurements at μs-level intervals will yield fairly uniform values, irrespective of the duration of the loop, as long as 1 μs includes a few iterations. However, IPC measurements at ms-level intervals will yield uniform values only if the loop lasts for many ms. In our applications, much of the code appears to exhibit more uniformity at μs-level intervals than at ms-level intervals. Consequently, we set the interval between measurements (microcycle) to a few μs and, therefore, implement the Slack algorithm in hardware.

2.4.2 Distribution vs Centralization

We now consider how to apply our framework to chips with multiple processor cores. Ideally, we would like to run the framework in a distributed manner. Each processor would have its own framework, running algorithms that read local sensors and make decisions on what techniques to activate locally. This approach is appealing because, potentially, each processor may be running a very different application.

In practice, while some energy-management techniques like those that modify the cache hierarchy can be easily controlled on a per-processor basis, other techniques are best controlled for the whole chip. Consider, for example, voltage-frequency scaling. Using a different voltage and frequency in each processor neighborhood introduces complexity and makes communication between the processors trickier.

One possible alternative is to use per-processor frameworks to run the algorithms and then, after a global synchronization step, make a global decision. However, such an approach is likely to suffer from synchronization overhead.

The approach that we take is to run the algorithm in a centralized manner. Signals from the different processor neighborhoods bring information from the distributed sensors to a central framework module. The module feeds the highest temperature and the sum of all the instructions executed to a centralized algorithm. While this approach requires a more careful timing design, it simplifies the decision-making process.

2.5 Energy Management Techniques

The different energy-management techniques in the framework will target different components of the chip and impact the energy, execution time (delay), and energy-delay product of applications differently. In this section, we select a few, representative techniques to include in the prototype framework that will be evaluated in Section 4.

All the techniques that we select reduce the average power consumption at the expense of slowing down the application. However, while some techniques reduce the total energy consumed in the application run, others do not. Consequently, the techniques in the first group may or may not decrease the energy-delay product, while those in the second group always increase it.

Among the techniques in the first group, we include: sub-banked data caches [9, 30], filter instruction caches [20], voltage-frequency scaling [11, 13], and reduced memory voltage [16]. In each of these cases, when the technique is activated, the system goes from a default configuration to a lower-energy, lower-performance one. These techniques can be used for both the Thermal and Slack algorithms.

Among the techniques in the second group, we include slowing down data cache hits and putting the processor to light sleep. These techniques simply introduce extra delay to reduce the average power. Due to their energy inefficiency, we will try to keep them out of our Thermal and Slack algorithms. However, they may contribute to the thermal crisis support.

We now briefly describe these techniques, while a more detailed description can be found in [36]. The values used for their parameters are listed in Section 3.1. Our framework can be easily extended to include other techniques.

Sub-Banked Data Cache

With cache sub-banking, a cache access activates only part of the cache line selected instead of the whole line [9, 30]. To support sub-banking, the cache is augmented with additional decoding logic and transmission gates. When sub-banking is not activated, this logic adds negligible delay to the cache access time.

When sub-banking is activated, a cache access consumes less energy. This is because the number of activated bit lines and sense amplifiers is reduced. However, the presence of the extra decoding logic and transmission gates tends to increase the cache access time. Consequently, cache hits consume less energy but are slower. The energy consumption and speed of cache misses are unaffected.

Filter Instruction Cache

The on-chip I-memories that supply instructions to the processors in an embedded chip are often designed with high-performance SRAM to ensure that their latency is minimal. They are also large, to hold the whole program. As a result, each access to them, while fast, consumes significant energy.

To address this problem, a small I-cache can be placed between the I-memory and the processor. Accesses to this cache are not faster in number of cycles than accesses to the already fast I-memory. However, they consume much less energy. As a result, this cache works somewhat like a filter cache [20].

If this filter cache is deactivated, all fetches go directly to memory, enabling a fast yet energy-consuming system. If, instead, the cache is activated, hits in the cache take the same time but consume much less energy. Misses, however, force the fetch to go to memory, adding up additional latency and energy consumption. Overall, with the cache activated, the system is likely to be slower but consume less energy.

An alternative design could be to eliminate the filter cache and add sub-banking to the I-memory. In such a design, however, accesses to an I-memory sub-bank could suffer one extra cycle of latency. The result is likely to be a slower system than the one with the filter cache.

Voltage-Frequency Scaling

Reducing both the voltage and the frequency of the chip is a well-known technique [11, 13]. Dynamic energy is proportional to the square of the supply voltage, while dynamic power is proportional to the frequency and to the square of the voltage. To apply this technique, we simply reduce linearly the voltage and frequency of the whole chip to V_{dd_low} and f_{low}. This change works for the linear section of the scaling curve.

Reduced Memory Voltage

We lower the voltage of only the DRAM array to V_{mem_low}. This can be done by changing the reference voltage used in an on-chip voltage converter according to the outputs of a detector [16]. Voltage changes have to be managed carefully because they induce non-linear changes to transistor characteristics. In this technique, to scale down other parameters as we scale down the voltage, we use circuit simulations. In addition, during the low-power mode, we also change the DRAM refresh intervals. The procedure that we use is outlined in [36].

Slowing Down Data Cache Hits

This technique progressively reduces the number of outstanding data loads and stores that a processor can have and, later, increases the latency of cache hits. More specifically, the number of allowed outstanding accesses is progressively halved. Once we reach 1 load and 1 store, we progressively increase the cache hit latency one cycle at a time. When this technique is to be deactivated, we undo these changes in reverse order.

Light Sleep Mode

In this technique, we put the processor in a light sleep mode for a period of time. We do not turn off the PLL, clock distribution, or DLLs to minimize any wake-up penalty. We simply gate the clock at the output of the DLLs. Since, by default, we were already clock-gating all the units not used, this technique cannot save much energy. In fact, because we are keeping the PLL, DLLs, and clock distribution lines on while slowing down the application, this technique ends up increasing the energy consumed. However, it reduces the average power consumed in the system.

3 EVALUATION ENVIRONMENT

We evaluate an implementation of our adaptive framework on top of an advanced chip with multiple superscalar cores and DRAM banks. We use detailed software simulations at the architectural level. The simulations are performed using

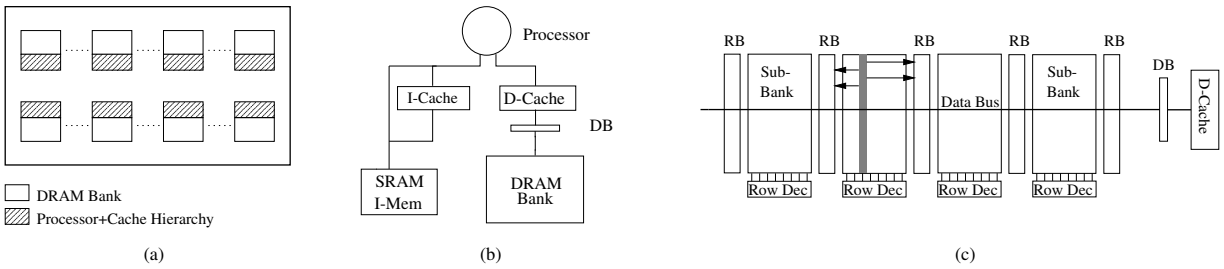


Figure 3: Chip architecture modeled: overview of the chip (a), per-processor memory hierarchy (b), and per-processor DRAM bank organization (c). In the charts, *RB*, *DB*, and *Row Dec* stand for row buffer, data buffer, and row decoder, respectively.

Processor	D-Cache	I-Cache	I-Memory	Data Buffer	Row Buffer	DRAM Sub-Bank
2-issue in-order at 800 MHz BR Penalty: 2 cycles Int,Ld/St,FP Units: 2,1,0 Pending Ld,St: 2,2	Size: 8 KB Assoc: 2 Line: 32 B RTrip: 1.25 ns	Size: 128 inst. Assoc: 1 Line: 4 inst. RTrip: 1.25 ns	Size: 8 KB Line: 4 inst. RTrip: 1.25 ns	Number: 1 Size: 256 b Bus: 256 b RTrip: 3.75 ns	Number: 5 Size: 1 KB Bus: 256 b RTrip: 7.5 ns	Number: 4 Num Cols: 4096 Num Rows: 512 RTrip: 15 ns

Table 1: Parameters for a single memory bank and processor pair. In the table, *BR* and *RTrip* stand for branch and contention-free round-trip latency from the processor, respectively.

Technique	Label	Parameter Value
Sub-banked data cache	*SubBank*	Cache hit if no sub-banking: RTrip = 1.25 ns, E = 222.8 pJ Cache hit if sub-banking: RTrip = 2.50 ns, E = 69.1 pJ
Filter instruction cache	*IFilter*	I-mem access: RTrip = 1.25 ns, E/inst = 51.6 pJ I-cache hit: RTrip = 1.25 ns, E/inst = 15.4 pJ I-cache miss + I-mem access: RTrip = 2.5 ns, E/inst = 67.0 pJ
Voltage-freq. scaling	*VoltFreq*	V_{dd_low} = 1.44 V, f_{low} = 640 MHz, overhead of any scaling = 10 μs
Reduced memory voltage	*MemVolt*	V_{dd} = 1.8 V: RB access (RTrip = 7.5 ns, E = 500.1 pJ), DRAM access (RTrip = 15 ns, E = 3702.2 pJ) V_{dd} = 1.2 V: RB access (RTrip = 7.5 ns, E = 500.1 pJ), DRAM access (RTrip = 21.25 ns, E = 2634.6 pJ)
Slowing D-cache hits	*SloHit*	–
Light sleep mode	*Sleep*	–

Table 2: Values of the parameters used in our energy-management techniques. In the table, *E*, *RB*, and *RTrip* stand for energy, row buffer, and contention-free round-trip latency from the processor, respectively.

a MINT-based [32] execution-driven simulation system [21] that models all the components of the chip, including the superscalar processors. The simulator includes energy consumption models. In the following, we describe the architecture modeled, how we estimate the energy consumed, the applications executed, and the metrics used.

3.1 Architecture Modeled

As an example of an advanced chip, we model a processor-in-memory chip with 64 simple processors cycling at 800 MHz and 64 Mbytes of DRAM. The target technology is IBM's 0.18 μm Blue Logic SA-27E ASIC [12] with some expected improvements in DRAM density [36]. The default voltage is 1.8 V.

The chip is modeled after a *FlexRAM* chip [19]. Processors are 2-issue wide and statically scheduled. Each processor is associated with a 1-Mbyte DRAM bank. A processor can directly access its own DRAM bank as well as the DRAM of its left and right neighbors. Such support allows communication between the processors, effectively connecting them in a ring. In addition, as in *FlexRAM*, the chip contains an on-chip controller that executes the serial sections of the application, including initialization, broadcast, and reduction operations [19]. The controller's contribution to the execution of our applications constitutes on average only 8% of the time, and is mostly limited to the initialization and ending parts of the application. For these reasons and because most chip resources are very underutilized when the controller runs, we do not include the controller's contribution in our evaluation.

Figure 3 shows the architecture of the chip. In the figure, Chart (a) gives an overview of the chip, while Chart (b) shows the memory hierarchy of each processor in the chip and Chart (c) shows the organization of each DRAM bank into sub-banks. Table 1 shows the most important architectural parameters for a single memory bank and processor pair.

Table 2 shows the values for the parameters of the energy-management techniques included in our framework. The energy values used will be justified in the next section. The values of some other framework parameters are as follows. Changing the memory voltage with *MemVolt* is assumed to have negligible overheard. Both the thermal and the slack macrocycles are set to 1 ms, while the microcycle is set to 1 μs. To avoid instability in the Thermal algorithm, we set a different *MinTemp* for each technique, as shown in Section 3.4. Finally, every time that we execute the Thermal algorithm, we charge 200 cycles to account for the overhead of the execution in the OS.

3.2 Estimating the Energy Consumed

To estimate the energy consumed in the chip, we have applied scaling-down theory to data on existing devices reported in the literature, as well as used several techniques and formulas reported in the literature [3, 30, 18, 24, 34, 35]. A detailed discussion of the methods that we have followed can be found in [36]. In this section, we give an overview of how we estimate the energy consumed in the processor cores, memory hierarchies, and clocks. We also discuss how we validated the models.

Processor Cores

Each core is a 32-bit 2-issue processor with a DLX-like pipeline. It supports a simplified version of the MIPS ISA with only 28 16-bit instructions [19]. We take the data from [35] and, by applying general scaling theory and considering technology trends, we estimate the average energy consumed in the register file, branch unit, ALU, and the other modules of the processor. Then, we can estimate the energy consumed by each type of instruction by adding up the energy of all the modules used by that particular instruction type. We assume perfect clock gating inside the processor code. With this approach, for example, we estimate that an add, a branch, and a multiply instruction consume an average of 56.1, 34.8, and 251.2 pJ, respectively.

Memory Hierarchies

To compute the energy consumed in the memory hierarchy, we use popular models [30, 18]. We classify memory hierarchy accesses based on what level of the hierarchy they reach, and depending on whether they are reads, writes, or dirty line displacements. Then, we compute the average energy consumed by one access of each class. This is done by dividing the access into simple operations. For example, a read that hits in the row buffer is divided into a cache tag check, a read hit in the row buffer, and a line fill into the cache. Finally, to compute the overall energy in the memory hierarchy, we multiply the number of accesses of each class times the corresponding energy per access in the class, and then accumulate the contribution of all classes. As an example, Table 3 shows the average energy consumed by a read and a write access to different levels of the hierarchy.

Level of the Hierarchy	Rd Energy (pJ)	Wr Energy (pJ)
D-cache	222.8	246.3
I-mem (per instr)	51.6	56.8
Row buffer	500.1	2740.6
DRAM bank	3702.2	3286.2

Table 3: Average energy consumption per access.

Clocks & Other

The clocking system includes 1 main PLL and 16 distributed local DLLs [29]. The clock network is laid out in the chip using an H-tree structure to minimize skew. To estimate the overall energy of the clocking system, we estimate and add the contributions of several components, namely PLL, DLLs, buffers, and distribution lines. Such contributions are estimated based on [3] and on capacitance models. Overall, the estimated average energy per cycle is 957.5 pJ. This figure does not include the energy for the clock inside the processor cores. The latter is included in the computation for the cores. Further details can be found in [36].

Validation

We validate our energy estimates with several experiments. We report on two of them here. In the first validation, we examine our cache model. We compare our energy estimates to those generated with the CACTI v2 models [34]. Since CACTI uses a relatively old sense amplifier model, we change it to a more aggressive one. The comparison shows that our estimates of energy consumption in the data cache and CACTI's are only 9% different [36].

In a second validation, we focus on the relative energy consumption of the I-cache, D-cache, clock, and processor core. Such a relative breakdown of energy for the Strong ARM processor is available from [24]. We compute the corresponding estimates for one of our processors plus its associated caches and share of the clock. While there are some differences between the two architectures, getting a similar breakdown is reassuring. The comparison shows that the contribution of each of the components does not differ by more than an absolute 6% between the two systems [36].

3.3 Applications Executed

For the experiments, we use 6 applications that are suitable to the integer-based processor-in-memory chip considered: they access a large memory size, are very parallel, and are integer based. They come from several industrial sources. We have parallelized each application into 64 threads by hand.

Table 4 lists the applications and their characteristics. They include the domains of data mining, neural networks, protein matching, multimedia, and image compression. Each application runs for several billions of instructions. Appendix A gives more information on each application.

3.4 Metrics Used

We characterize an application run with four metrics: performance (measured with total execution time, also called *delay*), average power consumption, total energy consumption, and product of energy times execution time (energy-delay product [10]). We will strive for a low energy-delay product, since it implies a good balance between high speed and low energy consumption.

In some experiments, we need to estimate chip temperature. However, our models only use energy and power metrics. We currently do not have a thermal model that, taking into account the chip package and cooling support, translates sustained power dissipation into chip temperature.

It is known, however, that heat transfers occur at the ms level [31]. As a result, it has been suggested to use the average power dissipated over many cycles as a proxy for temperature [5]. We follow this approach and use a metric called $Power^*$ as a proxy for chip temperature. At a given time, $Power^*$ is 0.75 times the average power consumed by the chip in the last millisecond plus 0.25 times the value of $Power^*$ a millisecond ago. While clearly not perfect, this recursive definition tries to approximate the behavior of temperature. Using this metric, the proxy for *MinTemp* for *VoltFreq*, *SubBank*, and *IFilter* is set to 45%, 75%, and 78%, respectively of the proxy for *MaxTemp*.

4 EVALUATING THE FRAMEWORK

To assess our DEETM framework, we evaluate three issues: the management of multiple energy-management techniques (Section 4.1), the Thermal algorithm (Section 4.2), and the Slack algorithm (Section 4.3).

4.1 Technique Analysis & Comparison

Given a DEETM framework with multiple techniques, the first question to ask is what combination of techniques should it apply and in what order. We now answer this question for our framework.

Comparing Individual Techniques

We start by comparing the individual techniques with the following experiment for each application. We execute the application without activating any technique and record the average power dissipated P_{orig} (last column of Table 4). Then, for each technique, we perform four runs dynamically activating the technique with different intensities. The inten-

Appl.	What It Does	Problem Size	D-Cache Hit Rate	Average Power(W)
GTree	Data mining: tree generation	5 MB database, 77.9 K records, 29 attributes/record	0.507	10.2
DTree	Data mining: tree deployment	1.5 MB database, 17.4 K records, 29 attributes/record	0.986	10.8
BSOM	BSOM neural network	2 K entries, 104 dimensions, 2 iterations, 16-node network, 832 KB database	0.947	15.5
BLAST	BLAST protein matching	12.3 K sequences, 4.1 MB total, 1 query of 317 bytes	0.969	8.7
Mpeg	MPEG-2 motion estimation	1 1024x256-pixel frame plus a reference frame. Total 512 KB	0.999	11.3
FIC	Fractal image compressor	1 512x512-pixel image, 4 512x512-pixel internal data structure. Total 2 MB	0.978	6.1

Table 4: Applications executed.

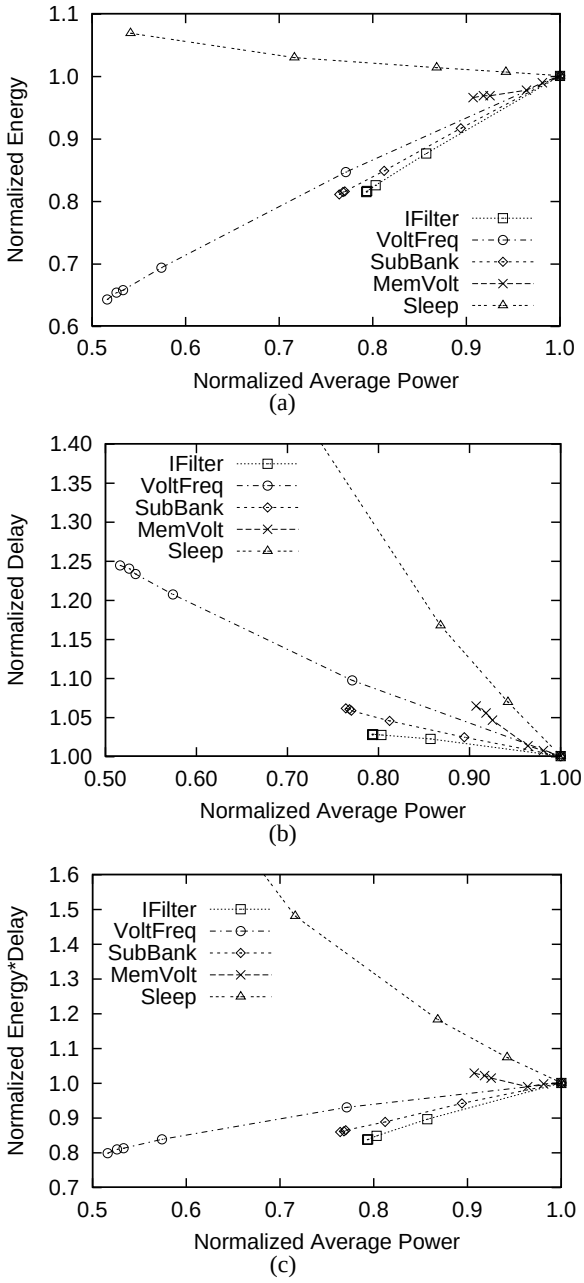


Figure 4: Impact of dynamically applying each individual energy-management technique: total energy consumed by the applications (a), their execution time (b), and their energy-delay product (c). The data is normalized to a run with no active technique and then averaged out across all applications.

sity is regulated with a power threshold: if the power in the last microcycle was over the threshold, the technique gets activated; the technique is deactivated when the power in the last microcycle was such that the technique could be deactivated without going over the threshold again. We set the thresholds to $1.2 \times P_{orig}$, $1.0 \times P_{orig}$, $0.8 \times P_{orig}$, and $0.6 \times P_{orig}$. Finally, we perform an experiment activating the technique for the whole run.

Figure 4 shows the results. The results of each run have been normalized to the run with no active technique for the same application, and then averaged out across all applications. The figure shows the resulting average power consumed in the run (X axes) against the total energy consumed (Chart (a)), execution time (Chart (b), where execution time is labeled *Delay*), and energy-delay product (Chart (c)). Since *SloHit* has a behavior very similar to *Sleep*, we do not show *SloHit* to simplify the charts.

The figure shows that the behavior of *Sleep* is different from the others as the average power decreases. *Sleep* does not reduce the energy (Chart (a)), substantially slows down the applications (Chart (b)) and, as a result, increases the energy-delay product significantly (Chart (c)). Consequently, due to its inefficiency, we only use it as the last resort in a thermal crisis.

The other four techniques (*IFilter*, *SubBank*, *VoltFreq*, and *MemVolt*) decrease the energy consumed by the chip (Chart (a)) and, while they still slow down the application (Chart (b)), they manage to reduce the energy-delay product or keep it roughly constant (Chart (c)). They differ significantly, however, in the slope of their curves and in the maximum power reduction that they can deliver. The maximum reduction is delivered when they are applied statically. This situation corresponds to the leftmost point of each curve.

To compare these four techniques to each other, we examine Chart (c). Recall that we want to minimize the energy-delay products. Under this requirement, the chart tells us what is the best technique to apply individually, and how to rank the techniques in case we want to apply them in a combined manner.

If we want to apply a single technique, we should choose the one that, for the desired average power reduction, delivers the lowest energy-delay product. For example, for power reductions that are less than 20%, *IFilter* is the best. *SubBank* is the best if we want reductions between 20 and 25%, while *VoltFreq* is the best for reductions larger than 25%. From this data, we can see that *IFilter* and *SubBank* are good but limited. Since their scope is only memory system accesses, they deliver modest power reductions.

If, instead, we want to rank the techniques for a possible combined application of them, what matters is not the absolute power reduction but the slope of the curves. Specifically, we approximate each curve with a straight line and record the slope of the line. The techniques with the highest positive slopes should be given the highest priority. Consequently, in our framework, the order of application of the techniques, irrespective of the power reduction desired, should be *IFilter*,

then *SubBank*, then *VoltFreq*, and so on.

Note that, for our techniques, the shape of the curves makes it possible to reasonably approximate each curve with a single straight line. This may not be true, however, in other scenarios, where we would need different straight lines in different segments of a given curve. In this case, the ranking of techniques would not be as straightforward: it would depend on the power reduction desired.

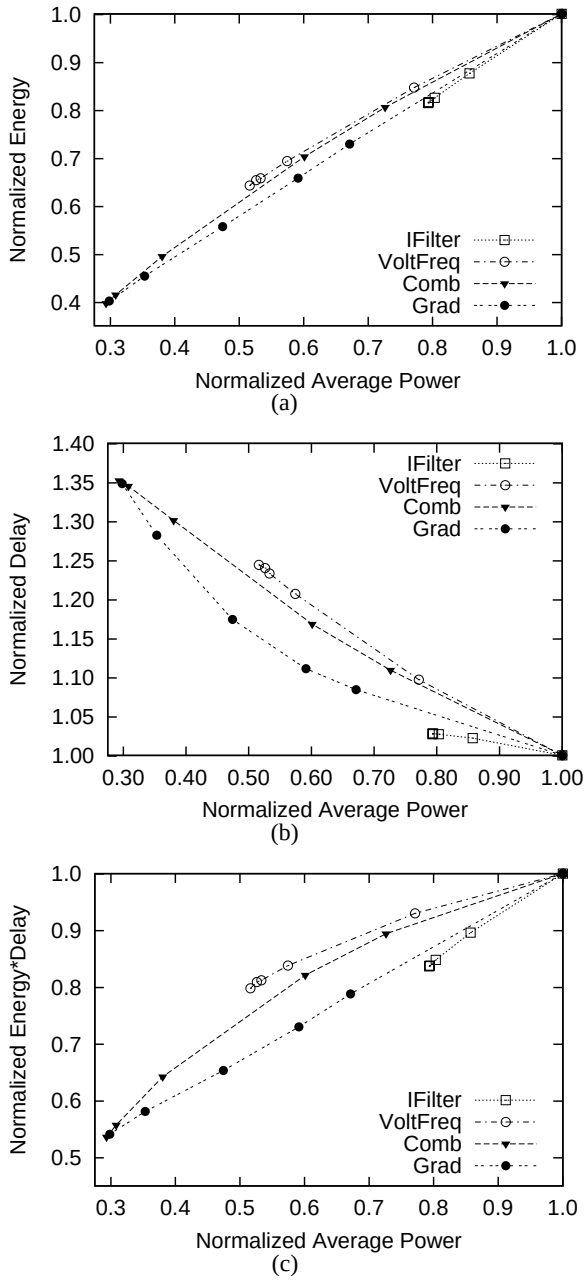


Figure 5: Impact of dynamically applying a combination of energy-management techniques: total energy consumed by the applications (a), their execution time (b), and their energy-delay product (c). The data is organized as in Figure 4.

Another complication occurs if the slope of a curve changes when the technique is combined with other techniques. While we have observed this effect in our framework, it does not change the ranking of techniques listed above.

Finally, we note that *MemVolt* reduces neither the average power much nor the energy-delay product. It is, therefore, unattractive. Its scope for impact is limited to applications with many cache misses. Unfortunately, even in this case, we find that it works poorly because the slower DRAM becomes a contention bottleneck that slows down the application (Chart (b)).

Applying Combined Schemes

To see the potential of our framework, we combine the three most effective techniques, namely *IFilter*, *SubBank*, and *VoltFreq*, into a single scheme. We consider two different schemes: *Comb* activates and deactivates the three techniques simultaneously, while *Grad* activates and deactivates them gradually. *Grad* uses the ranking selected before: it activates *IFilter* first; if more power or energy reduction is needed, it activates *SubBank*; if more is needed, it activates *VoltFreq*. When the techniques must be deactivated, it follows the reverse order.

Figure 5 shows the results of repeating the experiments of Figure 4 for *Comb* and *Grad*. For reference purposes, the figure also includes the curves for *VoltFreq* and *IFilter* from Figure 4. Note, however, that the axes have been expanded relative to Figure 4.

We can see from Figure 5 that, for modest power reductions, the effectiveness of *Comb* is between that of *VoltFreq* and *IFilter*. Specifically, Chart (c) shows that, for a given power reduction, the energy-delay product of *Comb* is between that of *VoltFreq* and *IFilter*. Consequently, *Comb* works well. In addition, *Comb* can deliver much higher power reductions than the individual techniques: if *Comb* is statically applied, it can reduce the average power by up to 70%. As a result, the final energy-delay product obtained in Chart (c) is also much lower than for the individual techniques.

As can be seen in the figure, however, *Grad* is better. Chart (c) shows that, for modest power reductions, this scheme delivers energy-delay products that are nearly as low as *IFilter*, the best of the three techniques. This is because, for this range of reductions, *Grad* is largely *IFilter*. When larger reductions are desired, *Grad* starts using the less optimal techniques. Finally, as we approach large reductions, it gets closer to *Comb*. In all cases except static application, however, *Grad* has a lower energy-delay product than *Comb* (Chart (c)).

These results form the rationale behind our choice of Thermal and Slack algorithms in Section 2.2: a gradual, priority-ordered application of techniques that reduce the energy-delay product. Consequently, we implement the Slack and Thermal algorithms with *Grad*. In addition, as part of the Thermal algorithm, we keep one additional technique ready for activation in case of a thermal crisis. Such a technique, which must be able to reduce the average power consumed as much as needed, is chosen to be *Sleep*.

Variation Across Applications

Finally, we note that, although different applications behave differently, the schemes chosen for our adaptive framework work well across all applications. For lack of space, we only briefly discuss the two individual applications that diverge the most from the average: *GTree* and *DTree*. *GTree* has a high data cache miss rate (Table 4), which causes *SubBank* to have relatively less impact. *DTree*, on the other hand, has relatively more I-cache misses, which causes *IFilter* to be less effective. Overall, however, it can be shown that *Grad* is very effective: it reduces the energy-delay product significantly, while enabling large reductions in average power.

4.2 Evaluating the Thermal Algorithm

The goal of the Thermal algorithm is to keep the temperature of the chip lower than *MaxTemp*, while minimizing any resulting application slowdown. In addition, under no condition should the temperature surpass *CrisisTemp*. As indicated before, we use *Grad* and, if *CrisisTemp* is reached, we activate *Sleep*. We call the resulting scheme *Grad+Sleep*.

To show that *Grad+Sleep* is effective, we demonstrate that, given different *MaxTemp* temperature limits, it effectively keeps the chip temperature below *MaxTemp* practically all the time, while slowing down the execution only modestly. Recall that, as stated in Section 3.4, we use $Power^*$ as a proxy for temperature.

In Figure 6, we show the results of applying *Grad+Sleep* under different $Power^*$ limits. These limits are proxies for *MaxTemp*. For each application, the limits considered are $1.2 \times P_{orig}$, $1.0 \times P_{orig}$, $0.8 \times P_{orig}$, and $0.6 \times P_{orig}$, where P_{orig} is the original average power of the application (last column of Table 4). To get an idea of the absolute values of these limits, if we average them out across all the applications, we get 12.5, 10.4, 8.3, and 6.3 W, respectively. The crisis $Power^*$ is set sufficiently high such that it is never reached. As usual, the data is normalized to the original conditions of the application and then averaged out across all applications.

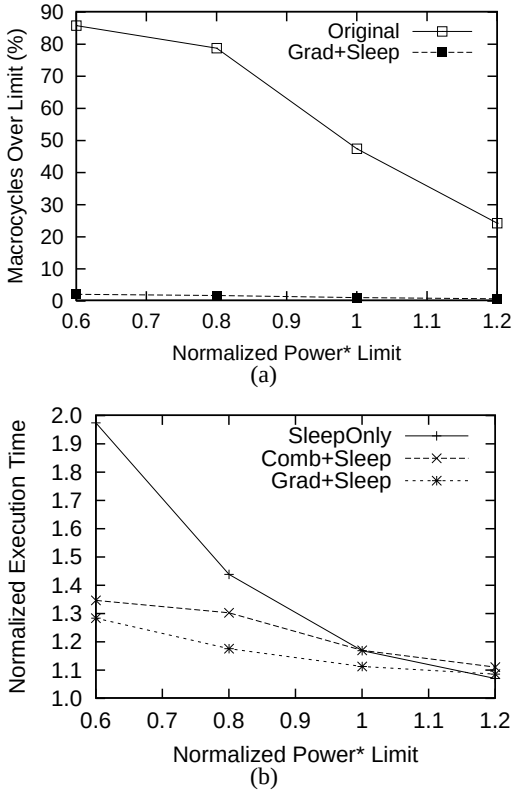


Figure 6: Impact of enforcing different $Power^*$ limits in the chip: fraction of thermal macrocycles over the $Power^*$ limit (a) and resulting execution time of the applications (b). These limits are proxies for *MaxTemp*.

Figure 6-(a) shows the fraction of thermal macrocycles where $Power^*$ is above the limit before we activate our framework (*Original*) and after (*Grad+Sleep*). The chart shows that, irrespective of how low we set the limit to, our framework keeps $Power^*$ below it for practically all the time. This is true even after setting the limit to 0.6 times the average power in the chip before activating the framework, which is the leftmost point of the chart. Such a limit places 85% of the *Original* macrocycles over the limit.

Figure 6-(b) shows the resulting execution time of the applications after activating the framework. If we focus on the *Grad+Sleep* curve, we see that, for modest limits, the scheme induces minimal slowdowns. For example, after setting the limit to 1.2 times the original average power, our framework only slows down the applications, on average, by 8%.

Overall, from the previous two discussions, we see that the goal of the Thermal algorithm is realized. For comparison purposes, however, Figure 6-(b) also shows the impact of using less efficient schemes. *Comb+Sleep* uses *Comb* instead of *Grad*. *SleepOnly* simply uses the *Sleep* technique when $Power^*$ surpasses the limit. More specifically, when a thermal macrocycle records a $Power^*$ higher than the limit, the fraction of non-sleeping cycles in the next macrocycle is decreased proportionally to how much $Power^*$ was over the limit. This scheme is, therefore, self-regulating. From the figure, we see that such schemes induce higher slowdowns than *Grad+Sleep*. *SleepOnly* is especially inefficient for relatively low $Power^*$ limits. However, it works well for the highest limit because it is being applied in a fine-grained manner.

4.3 Evaluating the Slack Algorithm

The goal of the Slack algorithm is to save as much energy as possible without extending the execution of the application beyond a given tolerable slack. As indicated before, we implement the algorithm with *Grad*. To show that our framework is effective, we demonstrate that, given different slack sizes, *Grad* delivers large energy savings without slowing down the job noticeably more than tolerable.

In Figure 7, the framework is tested with different slack sizes, specified as a percentage of the original execution time of the application. As usual, the data is normalized to the original conditions of the application and then averaged out across all applications.

Figure 7-(a) shows the resulting energy consumed by the applications for different slack sizes. The chart shows that *Grad* delivers large energy savings by exploiting even small slacks. For example, if the applications are allowed to exploit a 10% slack, they consume only 60% of the original energy; if they are given a 30% slack, they consume only 40%.

To put the effectiveness of *Grad* in perspective, the chart also shows the curves for $E \times D = constant$ and $E \times D^2 = constant$. As a reference, the voltage-frequency scaling technique [11, 13] often falls in between the $E \times D$ and $E \times D^2$ curves. Indeed, if the scaling of voltage and frequency is linear, since energy is proportional to the square of the voltage and delay is inversely proportional to the frequency, $E \times D^2$ remains constant. In practice, the scaling deviates from linear behavior and we move toward the $E \times D$ curve. Overall, from the distance between these curves and *Grad*, we can see that our framework is very effective, especially with small slacks.

Figure 7-(b) shows the fraction of the tolerable slack that is used up by our framework. We see that, for modest-sized slacks, *Grad* tends to deviate little from using the maximum tolerable slack. Any under- or over-utilization is limited to about 2% of the slack. As the slack increases over 35% of the execution time, the applications cannot use it all, even when all the techniques in *Grad* are in full operation. As a result, part of the slack is wasted. Overall, we see that the goal of the Slack algorithm is realized: *Grad* delivers large energy

reductions by exploiting even small slacks.

To gain insight into any possible improvements over *Grad*, Figures 7-(a) and 7-(b) also show the behavior of an ideal scheme that we call *Oracle*. At any given microcycle in the execution, *Oracle* applies the combination of *IFilter*, *SubBank*, and *VoltFreq* that best furthers the goal of the Slack algorithm. Since *Oracle* is based on perfect knowledge of the future, it should have, for a given slack, the lowest energy curve in Chart (a). In some cases, however, *Grad* reduces the energy slightly more than *Oracle*. This is because, due to imperfect prediction of the future, *Grad* sometimes goes slightly over the tolerable slack in Chart (b). Overall, however, the charts show that there is not much difference between the *Oracle* and *Grad* curves, which suggests that *Grad* is very competitive.

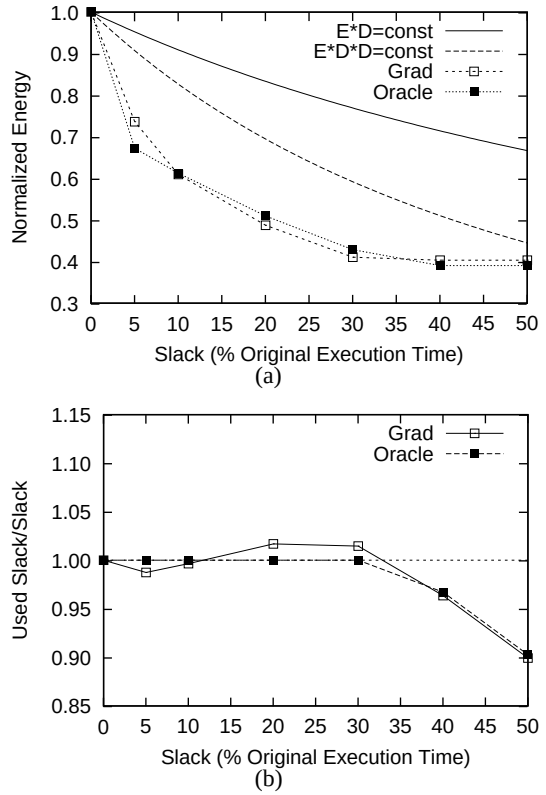


Figure 7: Effect of exploiting different execution slacks: resulting energy consumed by the applications (a) and fraction of the slack that is used up (b).

5 RELATED WORK

Of all the techniques and systems listed in Section 1, the work most related to ours is the one on dynamic systems for chip-level energy management. These systems can be classified into three groups. The first one targets temperature control, for example through context switching to jobs that consume less power [27] or through speculation control [5]. The second group targets energy efficiency without compromising performance, for instance through speculation control [23] or through reconfigurability [1]. A final group targets energy efficiency by exploiting slack and, therefore, slowing down the system. This is done, for example, through voltage and frequency scaling [25] or through switching to less aggressive instruction issue and speculation support [8]. Our work is different in two ways: we target both energy efficiency and temperature control, and we combine many techniques in a unified dynamic framework.

Recently, dynamic application of voltage and frequency scaling or various sleep modes have become popular among microprocessors [11, 13].

A related approach is that of ACPI (Advanced Configuration and Power Interface), an open industry specification that defines an interface for the OS to activate low-power modes [14]. Our work differs from ACPI in two ways. First, in ACPI, any decision and control of power modes is done by the OS. In our framework, the decision and control is best done with a combination of software and hardware, which enables finer-grained energy management. Second, current ACPI releases are only concerned with various sleeping modes, while we combine techniques that trade energy for performance.

ACPI and other OS-driven approaches have been used at the system level to save energy dynamically. For example, it is feasible to save energy by dynamically shutting down unused modules of the system like hard disks or the LAN [4]. Alternatively, the savings can come from dynamically reducing the quality of service to the application [7].

6 CONCLUSIONS AND FUTURE WORK

To address the problem of high energy consumption in current and upcoming chips, several schemes for dynamic energy management have recently been proposed. However, such schemes are still relatively limited and, in addition, tend to tackle only one of the two aspects of energy management: either energy efficiency or temperature control. To address these limitations, this paper has proposed a framework for Dynamic Energy Efficiency and Temperature Management (DEETM). The framework addresses the two aspects of energy management in a unified form. In addition, it combines a suite of energy-management techniques that can be activated individually or in groups according to a given policy.

The evaluation has shown that our framework is very effective, especially when the tolerable slowdowns and temperature limits are modest. In these scenarios, dynamic application of the most fitting techniques in the suite is most cost-effective: temperature limits are enforced with small slowdowns and large energy savings are delivered by exploiting small slacks. For example, the framework delivers a 40% energy reduction with only a 10% application slowdown. Overall, we feel that it makes sense for future advanced chips to include a DEETM framework like ours that combines multiple techniques.

As part of our ongoing work, we are trying to improve our DEETM framework by adding more techniques to it. We can then quantify the complementarity of and the overlap between different techniques.

Another approach that we are exploring is the potential of profiling. We can profile an application and, depending on what are its main energy and performance bottlenecks, tailor the activation of the techniques. Experience with the *Oracle* scheme in Section 4.3, however, suggests that little more can be done for the techniques and applications considered. However, other techniques and applications may behave differently. Finally, we are examining how to tailor the framework for different classes of chips, namely high-end microprocessors, chip multiprocessors, and different types of systems on a chip.

REFERENCES

[1] D. Albonesi. Dynamic IPC/Clock Rate Optimization. In *International Symposium on Computer Architecture*, pages 282–292, July 1998.

[2] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic Local Alignment Search Tool. *Journal of Molecular Biology*, 215(3):403–410, October 1990.

[3] J. Alvarez et al. A Wide-Bandwidth Low-Voltage PLL for PowerPC Microprocessors. *IEEE Journal on Solid-State Circuits*, 30(4):383–391, April 1995.

[4] L. Benini et al. Monitoring System Activity for OS-Directed Dynamic Power Management. In *International Symposium on Low Power Electronics and Design*, pages 185–190, August 1998.

[5] D. Brooks and M. Martonosi. Adaptive Thermal Management for High-Performance Microprocessors. In *Workshop on Complexity Effective Design*, June 2000.

[6] Y. Fisher. *Fractal Image Compression: Theory and Application*. Springer Verlag, 1995.

[7] J. Flinn and M. Satyanarayanan. Energy-Aware Adaptation for Mobile Applications. In *Symposium on Operating Systems Principles*, pages 48–63, December 1999.

[8] S. Ghiasi, J. Casmira, and D. Grunwald. Using IPC Variation in Workloads with Externally Specified Rates to Reduce Power Consumption. In *Workshop on Complexity-Effective Design*, June 2000.

[9] K. Ghose and M. Kamble. Reducing Power in Superscalar Processor Caches Using Subbanking, Multiple Line Buffers and Bit-Line Segmentation. In *International Symposium on Low Power Electronics and Design*, pages 70–75, August 1999.

[10] R. Gonzalez and M. Horowitz. Energy Dissipation In General Purpose Microprocessors. *IEEE Journal on Solid-State Circuits*, 31(4):1277–1284, September 1996.

[11] T. Halfhill. Transmeta Breaks x86 Low-Power Barrier. *Microprocessor Report*, 14(2):1,9–18, February 2000.

[12] IBM Microelectronics. Blue Logic SA-27E ASIC. http://www.chips.ibm. com/news/1999/sa27e/sa27e.pdf, February 1999.

[13] Intel. *Pentium III Processor Mobile Module: Mobile Module Connector 2 (MMC-2) Featuring Intel SpeedStep Technology*, 2000.

[14] Intel, Microsoft and Toshiba. *Advanced Configuration and Power Interface Specification*, 1999.

[15] K. Itoh. Low Power Memory Design. In *Low Power Design Methodologies*, pages 201–251. Kluwer Academic Publisher, 1996.

[16] K. Itoh et al. An Experimental 1Mb DRAM with On-Chip Voltage Limiter. In *ISSCC Digest of Technical Papers*, pages 84–85, February 1981.

[17] T. Juan, T. Lang, and J. Navarro. Reducing TLB Power Requirements. In *International Symposium on Low Power Electronics and Design*, pages 196–201, August 1997.

[18] M. Kamble and K. Ghose. Analytical Energy Dissipation Models for Low Power Caches. In *International Symposium on Low Power Electronics and Design*, pages 143–148, August 1997.

[19] Y. Kang, W. Huang, S. Yoo, D. Keen, Z. Ge, V. Lam, P. Pattnaik, and J. Torrellas. FlexRAM: Toward an Advanced Intelligent Memory System. In *International Conference on Computer Design*, pages 192–201, October 1999.

[20] J. Kin, M. Gupta, and W. Mangione-Smith. The Filter Cache: An Energy Efficient Memory Structure. *International Symposium on Microarchitecture*, pages 184–193, December 1997.

[21] V. Krishnan and J. Torrellas. An Execution-Driven Framework for Fast and Accurate Simulation of Superscalar Processors. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 286–293, October 1998.

[22] R. Lawrence, G. Almasi, and H. Rushmeier. A Scalable Parallel Algorithm for Self-Organizing Maps with Applications to Sparse Data Mining Problems. Technical report, IBM, January 1998.

[23] S. Manne, A. Klauser, and D. Grunwald. Pipeline Gating: Speculation Control for Energy Reduction. In *International Symposium on Computer Architecture*, pages 132–141, July 1998.

[24] J. Montanaro et al. A 160-MHz, 32-b, 0.5-W CMOS RISC Microprocessor. *IEEE Journal Solid State Circuits*, 31(11):1703–1714, November 1996.

[25] T. Pering, T. Burd, and R. Brodersen. The Simulation and Evaluation of Dynamic Voltage Scaling Algorithms. In *International Symposium on Low Power Electronics and Design*, pages 76–81, August 1998.

[26] J. Quinlan. *C4.5 - Programs for Machine Learning*. Morgan Kaufmann, 1993.

[27] E. Rohou and M. Smith. Dynamically Managing Processor Temperature and Power. In *2nd Workshop on Feedback-Directed Optimization*, November 1999.

[28] H. Sanchez et al. Thermal Management System for High Performance PowerPC Microprocessor. In *IEEE Computer Society International Conference*, pages 325–330, February 1997.

[29] S. Sidiropoulos and M. Horowitz. A Semidigital Dual Delay-Locked Loop. *IEEE Journal on Solid-state Circuits*, 32(11):1683–1692, November 1997.

[30] C-L. Su and A. Despain. Cache Design Trade-offs for Power and Performance Optimization: A Case Study. In *International Symposium on Low Power Electronics and Design*, pages 63–68, April 1995.

[31] C-H. Tsai. *Temperature-Aware VLSI Design and Analysis*. PhD thesis, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, May 2000.

[32] J. Veenstra and R. Fowler. MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. In *Second International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 201–207, January 1994.

[33] N. Vijaykrishnan et al. Energy-Driven Integrated Hardware-Software Optimizations Using SimplePower. In *International Symposium on Computer Architecture*, pages 95–106, June 2000.

[34] S. Wilton and N. Jouppi. CACTI: An Enhanced Cache Access and Cycle Time Model. *IEEE Journal on Solid-State Circuits*, 31(5):677–688, May 1996.

[35] N. Yeung. et al. The Design of a 55SPECint92 RISC Processor under 2W. *ISSCC Digest of Technical Papers*, pages 206–207, February 1994.

[36] S-M. Yoo, J. Renau, M. Huang, and J. Torrellas. FlexRAM Architecture Design Parameters. Technical Report CSRD-1584, Department of Computer Science, University of Illinois at Urbana-Champaign, October 2000. http://iacoma.cs.uiuc.edu/flexram/publications.html.

APPENDIX A: APPLICATIONS USED

This appendix describes the applications used. In the following, we use P.Mem to refer to the on-chip controller in the FlexRAM chip that executes the serial sections of the applications. More information on the applications can be found in [19].

GTree is a data mining application that generates a decision tree given a collection of records that we want to classify [26]. The records are distributed across the processors. The P.Mem decides what attributes to use to split the tree and tells the processors what branch they should examine. The processors process their records.

DTree uses the tree generated in *GTree* to classify a database of records [26]. Each processor has a copy of the decision tree and a portion of the database. Each processor processes its local records sequentially. At the end of the execution, the results are accumulated by P.Mem.

BSOM is a neural network that classifies data [22]. Each processor processes a portion of the input. Then, all processors synchronize, a summary of the partial results is combined and re-distributed, and the process begins again. While the original application used floating point, we have converted the application into fixed point to run on our simulated chip.

BLAST is a protein matching application [2]. The goal is to match an amino acid sequence sample against a large database of proteins. Each processor keeps a portion of the database and tries to match the sample against it. Finally, P.Mem gathers the results.

Mpeg performs MPEG-2 motion estimation. The reference image and the working image are distributed across the processors. Each 8x8 block in the working image is compared against the reference image.

FIC is a fractal image compression application that encodes an image using a scheme with a quad tree partition [6]. Each processor has a portion of the image and some calculated characteristics, and performs a local transformation to its portion of the image. The application may have significant load imbalance.