CAP: Criticality Analysis for Power-Efficient Speculative Multithreading*

James Tuck
NC State University
jtuck@ncsu.edu

Wei Liu
Intel Corp.
wei.w.liu@intel.com

Josep Torrellas
University of Illinois
torrellas@cs.uiuc.edu

Abstract

While Speculative Multithreading (SM) on a Chip Multiprocessor (CMP) has the ability to speed-up hard-to-parallelize applications, the power inefficiency of aggressive speculation is a concern. To improve SM's power efficiency, we note that not all the tasks that are running in a SM environment are equally critical.

To leverage this insight, this paper develops a novel, widely-applicable task-criticality model for SM. It also proposes CAP*, a novel architecture that builds a task-criticality graph dynamically and uses it to make scheduling decisions in a SM CMP. Experiments with SPECint, SPECfp, and Olden applications show that, in a CMP with one fast core and three slow ones, the $E \times D^2$ with CAP is, on average, 91–95% of that without. Moreover, it is only 77–91% of the $E \times D^2$ of a CMP with four fast cores and no CAP. Overall, we argue that scheduling for task criticality is beneficial.*

1. Introduction

Relentless transistor integration is driving processor manufacturers to build Chip Multiprocessor (CMP) architectures. However, while CMPs can effectively speed-up parallel programs, much of the application base today is still composed of sequential applications — for example, non-numerical applications that compilers fail to parallelize.

A proposed solution to speed-up these hard-to-parallelize codes is Speculative Multithreading (SM) (e.g., [5, 6, 16, 17]). While evaluations of SM on a CMP have generally shown good, if modest, speedups, an important concern has been the power inefficiency of aggressive speculation. As more tasks are executed speculatively to deliver higher speedups, there is a higher chance of spending power on work that ultimately gets squashed. Unfortunately, wasting power is an unattractive proposition.

Given the key importance of power issues, our goal is to design power-efficient SM systems. Previous work on this area by Renau *et al.* [12] focused on improving the energy efficiency of SM operations. In this paper, we focus instead on the interaction between the tasks of an application. Specifically, we make a *key observation* on the behavior of SM tasks: not all of the tasks that are running in a SM environment are equally critical for the performance of the application.

To leverage this insight to improve power efficiency, we need two architectural features. First, the CMP has to be able to assess the criticality of each task. While some of the ideas from instruction-level criticality analysis [4, 7, 14, 19, 20] can be reused for tasks, the model and hardware implementation required are substantially different. Second, the CMP has to be able to execute the less critical tasks in a way that saves power. This can be supported by running them on cores with lower frequency and voltage.

Based on these observations and needs, this paper proposes *CAP*, a novel architecture that (i) analyzes and predicts the criticality of tasks in a SM application at run-time, and (ii) uses criticality predictions to schedule tasks on a SM CMP with some cores running at lower frequency and voltage. Critical tasks are scheduled on fast cores, while non-critical ones run on slower ones.

Overall, this paper makes three contributions:

1. It develops a novel, widely-applicable task-criticality model for SM.
2. It proposes the CAP architecture, which dynamically builds a task-criticality graph based on the model and uses it to make scheduling decisions in a SM CMP.
3. It evaluates CAP. Experiments with SPECint, SPECfp, and Olden applications show that, in a CMP with one fast core and three slower ones, the $E \times D^2$ with CAP is, on average, 91–95% of that without it. Moreover, it is only 77–91% of the $E \times D^2$ of a CMP with four fast cores and no CAP. Overall, we argue that scheduling for task criticality is beneficial for power efficiency.

The rest of the paper is organized as follows. Section 2 presents a background; Sections 3 and 4 present the proposed criticality model and the CAP architecture; Sections 5 and 6 evaluate CAP; and Section 7 concludes.

2. Background

The critical path is the chain of dependent events that determine the overall execution time of the program. There

*This work was supported in part by the National Science Foundation under grant CCR-0325603; DARPA under grant NBCH30390004; DOE under grant B347886; SRC GRC under grant 2007-HJ-1592; and gifts from IBM and Intel.

has been substantial research on predicting the critical instructions and quantifying instruction criticality [2, 3, 4, 7, 14, 15, 19, 20]. Conventional instruction-level critical path analysis has the advantage of reasoning about events that happen within a microprocessor, such as contention on functional units. However, in the scenario of SM architectures, directly applying the instruction-level model has shortcomings. First, the flood of per-instruction information makes the model very hard to implement in hardware. Second, most per-instruction information is useless when considering interactions between tasks. Specifically, instructions that are not responsible for inter-thread communication are usually uninteresting. Finally, interesting, task-level information is scattered across the many instructions of the task.

Due to these limitations, in this paper, we propose a *task-level* criticality model for SM that tracks and collects *per-task* information, and then predicts the criticality of each task. The number of tasks is orders of magnitude lower than the number of instructions. Therefore, it is possible to store the model in hardware.

2.1 Speculative Multithreading (SM)

SM extracts tasks from a sequential program and executes them in parallel, hoping not to violate sequential semantics. The control flow of the sequential program imposes a task order. Therefore, we use the term predecessor and successor tasks. The safe (or non-speculative) task precedes all speculative tasks. SM schemes provide special hardware support to detect if the parallel execution of speculative tasks violates any data dependence relation required by the sequential program. If any dependence is violated, the offending tasks are squashed, any polluted state is repaired and the tasks are re-executed.

Buffering Speculative Data. Under SM, data generated by speculative tasks has to be stored separately before it can be merged with the safe state of the program. Speculative data can be saved in a special hardware buffer or in the cache of the processor. When a task becomes non-speculative, its data can be made visible to the rest of the system. A cache can hold speculative state from multiple speculative tasks. The data of each task has its own version ID.

In-Order and Out-of-Order Task Spawning. There are two types of task spawning schemes in SM: in-order and out-of-order [13]. Under in-order spawning, an individual task can at most spawn one correct task in its lifetime. A correct task is one that is in the sequential execution of the program, rather than in the wrong path of a branch. As a result, correct tasks are spawned in-order, namely, in the same order as in sequential execution. Under out-of-order spawning, an individual task can spawn multiple correct tasks. Out-of-order spawning is harder to support, but it enables more task parallelism.

Power-efficient Speculative Multithreading. Renau *et al.* [12] identified and quantified the main sources of energy consumption in SM. Then, they proposed a set of simple energy-saving optimizations for SM. Their work focuses on improving the efficiency of SM operations; our paper focuses on improving the efficiency of the interaction between SM tasks.

Nagpal and Bhowmik [10] used an instruction-level criticality model to drive energy-aware speculation for SM processors. They focused on controlling task squashes by delaying the non-critical loads and enhancing the branch prediction. Our scheme considers all task interactions, not just task squashes.

3. Task-Level Criticality Model

This section describes a novel task-level criticality model for SM environments. The model is very general, as it applies to both SMT- and CMP-based SM architectures, both with in-order and out-of-order task spawning. Since it tracks only task-level information, it significantly reduces storage requirements compared to an instruction-level scheme, and lends itself to a simpler hardware implementation.

3.1 Lifetime of a SM Task

In an instruction-level criticality model, an instruction is represented by a set of nodes that represent different stages in the instruction's lifetime — e.g., Dispatch, Execute and Retire. Similarly, a task experiences a few stages during its lifetime: Start (the task is created by a predecessor), Execute (the task is assigned to a free core or context), Finish (the execution completes the task-end instruction) and Commit (the commit token for the task is consumed). Figure 1(a) shows a task's lifetime, where the thicker line represents the use of a core or context.

To accurately calculate the critical path in a set of dynamic SM tasks, we need to model all the interactions between tasks: (i) a task can execute a spawn instruction, thereby creating another task; (ii) a task can execute a store instruction that synchronizes with a successor task or squashes it, forcing the latter and its successors to re-start; (iii) a task may have to wait for another to finish execution and release the core or context; and (iv) a task may have to wait for another to commit, so that it can become safe and either commit or perform an irreversible operation such as I/O.

We propose to divide a task lifetime into stages between which these interactions take place. Unlike the instruction-level model which has a unique set of nodes per instruction, many instructions map into a single node. As a result, using the four obvious stages Start, Execute, Finish and Commit as in Figure 1(a) is suboptimal. The reason is that the Execute stage can contain instructions with a *mix* of criticality levels. Specifically, if it contains a task spawn instruction (or similarly for a squash or sync), the instructions before

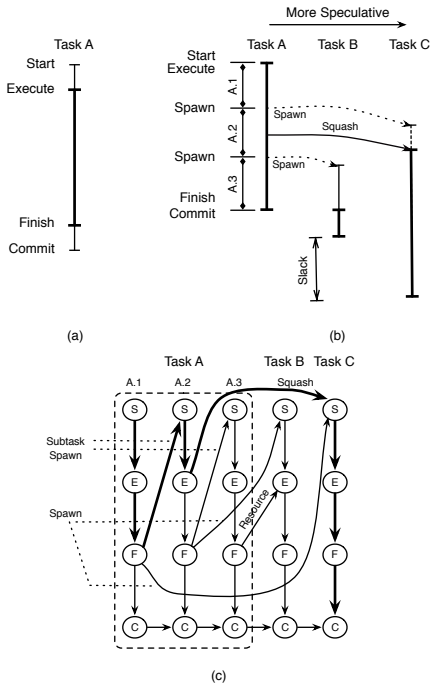

Figure 1. Stages in a task lifetime (a), example of multiple task execution (b), and resulting criticality graph (c).

the spawn may be much more or less critical than the ones after it. While the model accurately calculates the critical path in aggregate, its suboptimal characteristic makes it less accurate for identifying the criticality of a specific instruction.

3.2 Proposed Task-Level Model

In our model, we handle task spawn, squash, and synchronization operations in the Execute stage differently. Consider spawns first. In an environment with out-of-order spawning, a task may spawn multiple tasks. After every spawn operation, the criticality level of the task may change substantially. Consequently, we propose a model where a task is broken into *subtasks*. After every spawn operation, a new subtask starts.

As an example, consider Figure 1(b). It shows Task *A* spawning Task *C* and then squashing it. *C* immediately restarts. Later, Task *A* spawns Task *B*, which is less speculative than Task *C*. Task *B* cannot get a processor and waits until Task *A* finishes and relinquishes its processor. After that, Task *B* executes. When *B* completes, *C* is still running. In this example, there are three subtasks in *A*: from the start of the task to the first spawn (subtask *A.1*); between the two spawns (subtask *A.2*); and from the last spawn to the end of the task (subtask *A.3*).

This approach, where a subtask has at most one spawn, has two important properties. First, in-order and out-of-order task spawning environments are handled seamlessly. Secondly, a task spawn cannot cause instructions with different criticality levels to be included in the same Execute node. We could apply the same approach of creating a subtask for a squash or a synchronization edge. However, because they are much rarer than spawns, the benefits would be small.

We model a subtask with four nodes. Start (*S*) and Execute (*E*) follow the description of Section 3.1. Finish (*F*) corresponds to the point after the subtask executes a spawn or a task-end instruction. Finally, Commit (*C*) involves receiving the commit token, committing the architectural state of the whole task (only if this is the last subtask in the task), and passing the token to the next subtask. Table 1 shows when each node is reached.

Node	When It Is Reached
Start (*S*)	Subtask is created
Execute (*E*)	Subtask starts executing
Finish (*F*)	Subtask has completed a spawn or a task-end instruction
Commit (*C*)	Subtask receives the commit token

Table 1. Nodes in a subtask.

Edge	Description	
$S_i \rightarrow E_i$ $E_i \rightarrow F_i$ $F_i \rightarrow C_i$	Transitional edges within a subtask	
$F_i \rightarrow S_j$	*Spawn*	Subtask(i) spawns Subtask(j)
$E_i \rightarrow S_j$	*Squash*	Subtask(i) squashes Subtask(j)
$E_i \rightarrow E_j$	*Sync*	Subtask(i) synchronizes with Subtask(j)
$F_i \rightarrow E_j$	*Resource*	Subtask(i) relinquishes core to Subtask(j)
$C_i \rightarrow E_{i+1}$	*BeSafe*	Subtask(i+1) must wait to be safe
$C_i \rightarrow C_{i+1}$	*Commit*	Subtask(i)'s commit precedes Subtask(i+1)'s

Table 2. Edges in the criticality graph.

Table 2 describes all the possible edges between the nodes. The first set of edges are those within a subtask (Row 1). They are drawn between the successive stages of a subtask: $S_i \rightarrow E_i$, $E_i \rightarrow F_i$, and $F_i \rightarrow C_i$. The rest of the edges in the table are between subtasks, and represent interactions between subtasks. Specifically, every subtask starts in one of two ways. First, it can be spawned by a predecessor, as shown by the *Spawn* edge $F_i \rightarrow S_j$, where the subtask versions satisfy $i < j$ (Row 2). Alternately, the subtask can be squashed by a predecessor and restarted, as shown by the *Squash* edge $E_i \rightarrow S_j$ (Row 3).

Once executing, subtasks can synchronize either with explicit wait instructions or as a result of a dependence predictor. In this case, a *Sync* edge $E_i \rightarrow E_j$ is inserted (Row 4).

A spawned task may be unable to execute in two cases: it does not have a core or context to run on, or it needs to become safe before it can execute. The latter case may occur, for example, if the subtask has to perform an I/O operation, which cannot be executed speculatively. The first case is represented by the *Resource* edge $F_i \rightarrow E_j$ (Row 5). The second case is represented by the *BeSafe* edge $C_i \rightarrow E_{i+1}$ (Row 6). This edge goes from the Commit node of a subtask to the Execute node of its immediate successor.

Finally, the fact that each subtask must commit after its immediate predecessor commits is represented by the *Com-*

mit edge $C_i \rightarrow C_{i+1}$ (Row 7).

Figure 1(c) shows the criticality graph corresponding to the execution in Figure 1(b). The critical path can be computed efficiently using Last Arriving Rules [4] by walking backward from the last *Commit* node in the graph. The critical path in Figure 1(c) is highlighted with thicker lines. From this graph, it is clear that *Task B* is not in the critical path. *Task B* could take longer to execute (perhaps in a lower power mode) without hurting performance.

For simplicity, in the remainder of the paper, we will use the term "tasks" to mean the "subtasks."

4. Architecture Design

We propose an architecture called *CAP* (Criticality Analysis for Power-efficient Speculative Multithreading) to: (i) build our task-level criticality graph in hardware dynamically and (ii) make task criticality predictions based on the graph. These predictions are used to schedule SM tasks in a CMP power-efficiently.

Figure 2(a) shows a multiprocessor system with CAP. CAP adds two modules: a *Task Controller* (TC) and a *Critical Path Builder and Predictor* (CPBP). The TC keeps track of all running and pending tasks, and controls task scheduling. The CPBP has two components, namely the *Builder* and the *Predictor*. The former builds the criticality graph in hardware on-the-fly and computes the critical path; the latter uses the critical path information to store the inter-task edges that are *Critical Edges* most frequently. A Critical Edge is one that connects a critical task at the point where it ceases to be critical to the next critical task at the point where it starts being critical.

As a task executes, the TC keeps information on it, such as which task spawned it and which tasks squashed it. When the task commits, the TC forwards this information to the Builder to be added to the criticality graph. When the Builder has buffered enough tasks for a meaningful analysis, it calculates the critical path. Based on the critical path, the Builder updates the list of Critical Edges in the predictor. Then, the Builder's graph is flushed, and it starts over. The Predictor, therefore, gets continuously trained. The TC uses the information on Critical Edges in the Predictor to make scheduling decisions.

4.1 Task Controller

To be able to construct critical paths, we need to record some information for each dynamic task. Specifically, for each node in the task, we need to record (i) which incoming edge is the Last Arriving Edge (LAE) to the node and (ii) what is the ID of task at the source of that edge. This information is collected by the TC.

As a task is created, the TC assigns a Version ID (VID) to it — a unique name for that dynamic task. As the task executes, the TC records the aforementioned information.

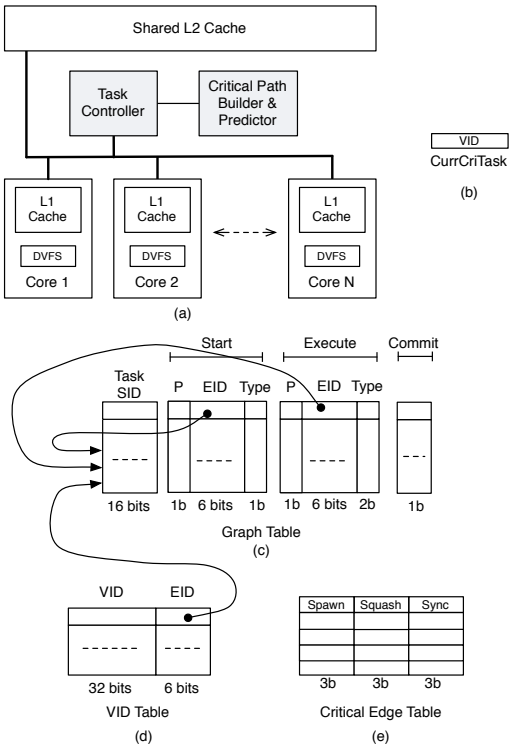

Figure 2. Multiprocessor system with CAP (a), CurCriTask register (b), Graph Table (c), VID Table (d), and Critical Edge Table (e).

When the task commits, this information is passed to the Builder.

The TC has a register called *CurCriTask* (Figure 2(b)), which contains the VID of the task that is currently critical. Such task is scheduled on a high performance core — one with high voltage and frequency. The other tasks can be scheduled on slower processors that use less power. Since such tasks had slack, they will not hurt performance. Every time that the CurCriTask task experiences an event that could be the source of a Critical Edge, the TC checks the Predictor. If the Predictor confirms that this is a Critical Edge with high probability, the task at the destination of the Critical Edge is placed in the CurCriTask register and the TC moves that task to a high-performance core.

Based on our experiments, we find that the Critical Edges where a processor re-schedule is most often beneficial are of the Spawn, Squash, and Sync type. Consequently, the Predictor will only store information on these three edge types. Moreover, the TC will only check the Predictor when the CurCriTask task spawns, squashes or synchronizes. If the Predictor has not recorded the corresponding Spawn, Squash, or Sync edge as Critical, criticality is assumed to remain in the task. In this case, the CurCriTask register remains unmodified.

However, recall that, in our model, a spawn also 'transforms' the spawning task into a new task — e.g., Task *A.1*

becomes *A.2* after spawning *C* in Figure 1. As a result, *A.1* and *A.2* have different VIDs. If the *A.1*→*C* edge is not found in the Predictor, the CurCriTask register is automatically updated to hold *A.2*. Artificial spawn edges introduced by our model such as *A.1*→*A.2* are not stored in the Predictor to save space.

When the CurCriTask task really finishes (i.e., not because of a spawn), the current non-speculative task in the system is moved to a high-performance processor, but the CurCriTask register is not necessarily updated. It is updated only if there is only one task left in the system — in which case, that task is put in the CurCriTask register. Otherwise, it is left unused and the TC checks the Predictor at every spawn, squash, and synchronization of every task, hoping to identify a Critical Edge. When one is identified, the destination task is put in the CurCriTask register.

The TC ensures that the non-speculative task always runs.

4.2 Builder

The Builder is responsible for building the graph and calculating the critical path. The Builder consists of the two hardware tables shown in Figures 2(c) and (d): the *Graph Table* and the *Version ID (VID) Table*, respectively. The Graph Table provides the storage for the criticality graph. The VID Table simply maps a VID to an index into the Graph Table — also called an Entry ID (EID).

Each row in the Graph Table corresponds to a dynamic task. The first field (*Task SID*) is the static task ID. The other fields record, for each node in the task, (i) the type of its LAE and (ii) the EID of the task at the source of its LAE. The different types of edges are shown in Table 2.

Specifically, the *Start* area of the Graph Table contains information on the LAE for the Start node. Since the alternatives are a Spawn or a Squash edge (Table 2), we use a 1-bit *Type*-of-edge field and an *EID* field. We discuss the *P* field later.

The *Execute* area contains information on the LAE for the Execute node. The alternatives are an intra-task edge from the task's Start node, or a Sync, Resource, or BeSafe edge. The first case is encoded with a null *EID* field. The other cases are encoded with a 2-bit *Type* field and the correct EID in the *EID* field.

There is no need to store information on the Finish node because it has a single type of incoming edge. Finally, the *Commit* area contains information on the LAE for the Commit node. The alternatives are an intra-task edge or a Commit edge. Consequently, we need a 1-bit *Type* field. There is no need for an EID field because the source of a Commit edge is always the immediate predecessor task.

The Graph Table also contains two *P* fields (Figure 2(c)) to help the Predictor. The *P* fields record whether the corresponding LAEs were predicted as critical during the previous execution of the task. By recording this information in the Graph Table, the Builder can compare the current critical path to criticality predictions made at the previous execution, and update the Predictor.

With this support, consider how a graph is built. When a task commits, an entry in the Graph Table is allocated for the task and is filled with information from the TC. The VID Table is used to map VIDs to entries in the Graph Table. When the Graph Table is filled with tasks, the critical path is calculated. A finite state machine traverses the graph in reverse and constructs the critical path based on LAE information. The cross-task edges in the critical path that are of Spawn, Squash or Sync type are Critical Edges and are passed to the Predictor for training.

Thanks to using a task-level criticality model, the Builder is very space-efficient. The storage for the Builder is 34b×64 for the Graph Table plus 38b×16 for the VID Table, for a total of 348 bytes.

4.3 Predictor

The Predictor records the most frequently-observed Critical Edges in the program. For that, it uses the Critical Edge Table (Figure 2(e)). Each row in this hardware table corresponds to an ordered pair of tasks that have one or more Critical Edges going between them. Each row contains three 3-bit up/down saturating counters, which count the number of times that we have seen a Critical Edge between these two tasks of type Spawn, Squash, and Sync, respectively.

In our model, a task 'transforms' itself into a new one when it spawns — e.g., Task *A.1* becomes *A.2* after spawning *C* in Figure 1. In the example, if *A.1* is critical, and the Critical Edge Table does not contain the *A.1*→*C* edge, we naturally assume that *A.2* is now critical. To save space in the Critical Edge Table, artificial spawn edges introduced by our model such as *A.1*→*A.2* are not stored in the table.

The table is trained every time that the Builder generates a critical path and identifies Critical Edges. For each Critical Edge, the static task IDs (Task SIDs) of the source and destination tasks are hashed into an index into the table. If no entry exists, a new one is allocated. In any case, the counter of the corresponding Critical Edge type is incremented. Note that, to improve the accuracy of the Predictor, we can use a more complex hashing function that incorporates additional path history — to distinguish between different calls to this pair of tasks.

This table is de-trained in two cases. The first one is when, after a critical path is generated, the Builder sees that a LAE with the *P* bit set (Section 4.2) is not part of the critical path. In this case, the corresponding counter in the Critical Edge Table is decremented. If, as a result, all counters in the row reach zero, the row is deallocated.

The second case is when an edge was predicted as critical during the previous execution of the task but the edge

did not even make it to the Graph Table because, in this execution, it was not even LAE. In this case, as soon the TC fills the corresponding row of the Graph Table, it decrements the corresponding counter in the Critical Edge Table.

With this support, the Critical Edge Table keeps the most important Critical Edges. The TC makes processor scheduling decisions based on the counter values in the table.

4.4 An Example of CAP at Work

Figure 3 provides an example of CAP at work. Parts (a) and (b) are the same as in Figure 1, except that we also show the VID Table. Consider each task in (a) as it commits. The first one to commit is Task *A.1*, which has a VID of 1. It is allocated at entry *EID*=29 of the Graph Table. This row is filled as follows. Since this is the first task, its Spawn and Execute nodes have no external edge. Hence, both the *Start* and *Execute* areas of the row are set to null. The *Commit* field is set to indicate that the LAE comes from the Finish node.

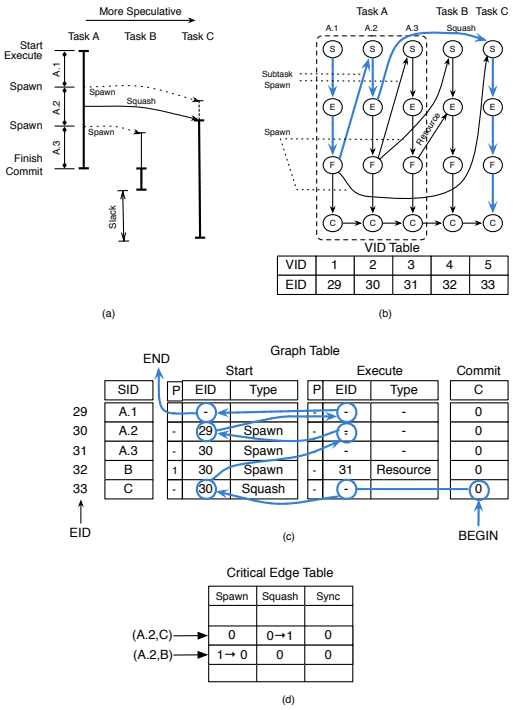

Figure 3. An example of CAP at work.

The next task to commit is Task *A.2*. It is given entry *EID*=30 in the Graph Table. It is spawned by Task *A.1*, which is the source of the LAE to the Start node. Consequently, the *Start* area of the row is filled with *EID*=29 and Spawn type. The *Execute* area is set to null because there are no external edges to the Execute node. The *Commit* field is set like in the previous task.

Task *A.3* proceeds in a similar fashion.

Task *B* commits next. The LAE to its Start node is a Spawn edge from Task *A.2*. Assume that such edge was a Critical Edge in the execution — hence *P*=1. The LAE to its Execute node is a Resource edge from Task *A.3*. The LAE to its Commit node comes from its Finish node.

Finally, Task *C* commits. The LAE to its Start node is a Squash edge from Task *A.2*.

The critical path is calculated by traversing the table in reverse, starting at the last committed task at the point marked by *BEGIN* in the figure. For each node, the Builder proceeds backward along LAEs. The dark lines with arrows drawn over the table show the progression of the critical path. For this traversal, the VID Table is not needed.

This critical path uncovers one Critical Edge that is added to the Critical Edge Table, namely the Squash edge between Tasks *A.2* and *C* [1]. If we assume that the corresponding rows in the Critical Edge Table contained all zeros, Figure 3(d) shows the resulting counter values. Moreover, we need to de-train the Spawn edge between Tasks *A.2* and *B*, which had *P*=1 and is not in the critical path. If the Critical Edge Table had a count equal to 1 for that edge, it now becomes zero.

5. Methodology

To evaluate CAP on a SM CMP, we use SESC [11] — a cycle-accurate execution-driven simulator with detailed models of out-of-order superscalars, memory subsystems, and SM protocols. It also includes models of dynamic power from Wattch [1], Orion [21] and Cacti [18]. Moreover, we augment the simulator to support per-core DVFS.

We consider two configurations of a CMP with SM support and three scheduling algorithms. One configuration is a 4-core CMP where each core runs at a high frequency (f) and voltage (V). We call it *4H*. The second configuration is the same 4-core CMP where only one core runs at the high f and V, while the other three cores run at a low f and V. We call it *1H*. For each configuration, we consider three scheduling algorithms, namely *Base*, *Sort*, and *CAP*. *Base* is a naive algorithm where tasks get free cores using first-come first-serve. *Sort* guarantees that the non-speculative task and the three least-speculative ones are always running; moreover, in *1H*, the non-speculative task runs on the sole fast core. *Sort* is based on the common wisdom that non- or less-speculative tasks are more important than more-speculative ones. Finally, *CAP* uses our proposed scheme. In all cases, we apply the energy optimizations for SM suggested by Renau *et al.* [12], including clock gating unused cores. Table 3 shows the architecture parameters.

To generate the SM binaries of applications, we use the POSH compiler [8]. We allow out-of-order spawn [13] and optimize the binaries for energy-efficiency [12]. We evaluate the SPECint and SPECfp 2000 applications with the *Ref* input set. Exceptions include those applications written

[1] Recall from Section 4.3 that artificial edges such as the Spawn edge between Tasks *A.1* and *A.2* are never added to the table.

Suite	Appl.	# Dyn. Tasks	# Inter-Task Edges/Task (%)				# Inter-Task Edges in Crit. Path (% of Each Type)						Similarity to Safe (%)
			Sq	Sync	Res	BSafe	Sp	Sq	Sync	Res	Com	BSafe	
Olden	bh	20464	2.0	0.0	97.4	1.5	0.7	2.0	0.0	95.4	0.6	1.3	25
	em3d	79874	0.9	99.0	81.9	0.0	18.0	2.3	0.3	79.3	0.0	0.0	2
	health	409490	0.1	0.0	32.5	29.3	47.1	0.0	0.0	16.5	5.2	31.1	24
	mst	1564246	0.2	0.0	54.4	0.0	47.2	0.4	0.0	52.2	0.0	0.0	19
	perimeter	51745	1.1	0.5	55.1	0.0	55.4	2.5	0.5	41.6	0.0	0.0	22
SPECfp	art	1727610	1.0	0.0	93.2	0.9	4.1	1.4	0.0	93.3	0.2	1.0	17
	equake	450582	1.8	1.2	88.6	0.7	23.4	2.9	0.5	72.5	0.3	0.4	16
	mesa	215426	0.0	100.0	10.4	0.0	19.9	0.0	77.7	2.3	0.0	0.0	31
SPECint	bzip2	981836	6.4	8.4	6.5	0.3	88.1	4.9	2.9	3.7	0.0	0.4	88
	crafty	665658	26.1	79.2	1.0	0.0	46.9	15.4	37.2	0.5	0.0	0.0	57
	gap	1335200	34.5	100.0	3.5	18.3	35.9	17.6	27.7	0.8	0.0	17.9	59
	gzip	2674412	0.8	41.6	0.1	0.0	79.6	0.7	19.6	0.1	0.0	0.0	38
	mcf	18607165	1.9	45.6	3.5	0.0	70.0	1.7	26.7	1.6	0.0	0.0	25
	parser	5261992	16.6	61.2	8.8	1.0	67.9	10.9	15.9	4.1	0.0	1.1	51
	twolf	6191552	18.1	43.2	6.3	0.0	37.4	20.2	38.8	3.6	0.0	0.0	39
	vortex	1838191	0.7	18.1	16.4	1.6	75.0	0.9	9.8	11.2	0.8	2.3	42
	vpr	174129	52.0	29.1	30.8	1.2	26.3	42.3	13.5	16.2	0.2	1.5	38
Olden Geom. Mean		140204	0.5	0.0	60.0	0.0	17.3	0.0	0.0	48.6	0.0	0.0	15
SPECfp Geom. Mean		551449	0.0	0.0	44.1	0.0	12.4	0.0	0.0	25.0	0.0	0.0	21
SPECint Geom. Mean		1972905	7.7	38.3	3.9	0.0	54.4	6.2	17.1	2.0	0.0	0.0	46

Table 4. Characterization of the critical path on *4H-Base*.

Processor	
Fetch/issue/comm width: 5/3/3	Frequency: 5.0 GHz
I-window/ROB size: 68/126	Branch predictor (spec. update):
Int/FP registers: 90/68	bimodal size: 16K entries
LdSt/Int/FP units: 1/2/1	gshare-11 size: 16K entries
Ld/St queue entries: 48/42	Cross-task dependence predictor:
Branch penalty: 13 cyc (min)	Pred. Table: 64 entries
BTB: 4K entries, 2-way assoc.	Sync. Table: 16 entries

CAP Builder	CAP Predictor
Graph Table: 64 entries	Critical Edge Table: 2048 entries
VID Table: 16 entries	

Caches (All line sizes:64B)	D-L1	I-L1	Shared L2
Size:	32KB	32KB	2MB
Round trip:	3 cyc	2 cyc	10 cyc
Assoc:	4	2	8

High frequency and voltage: 5 GHz and 1.6 V
Low frequency and voltage: 3.5 GHz and 1.1 V
Bus frequency: 533MHz; Bus width: 128bit
DDR-2 DRAM bandwidth: 8.528GB/s; memory round trip: 98ns

Table 3. Architecture parameters, where cycle counts are in processor cycles.

in C++ or Fortran, which we do not support, and *perlbmk* and *gcc* which fail in our compiler. We also evaluate a few Olden applications. In our simulations, we skip the initialization (1-6 billion instructions), and then execute about 0.75-1.50 billion sequential instructions.

For the SPECint applications, we studied the impact of a cross-task dependence predictor (Table 3) as described by Moshovos *et al.* [9]. Without the predictor, the average speedup of SM execution over sequential execution on the same platform is 1.32; with the predictor, the average speedup decreases to 1.28, but the $E \times D^2$ of the architecture decreases by 10%. Consequently, in all our experiments, we use the dependence predictor.

6. Evaluation

6.1 Characterization of Critical Paths

Table 4 characterizes the critical path of applications running on *4H-Base*. To provide more insight, the data ignores subtasks: all subtasks in a task are logically lumped into a single unit, and all edges between subtasks in the same task are neglected.

The third column shows the number of dynamic tasks per application. The group of columns labeled *Inter-Task Edges/Task* shows the number of inter-task edges of each type as a percentage of the number of tasks. Spawn and Commit edges are not shown because their number is 100%. From the table, we see that Squash and Sync edges are common in SPECint codes, while Resource edges are common in the SPECfp and Olden codes due to their higher parallelism. Having many Resource edges indicates that useful work is often waiting to execute. BeSafe edges are rare.

The group of columns labeled *Inter-Task Edges in Crit. Path* shows the percentage of inter-task edges in the critical path that are of each type. We see that the dominant types of Critical Edges are Spawn in SPECint and Resource in Olden and SPECfp. SPECint codes also have a substantial fraction of Squash and Sync Critical Edges.

The last column of the table shows the similarity between the dynamic instructions in the critical path and those in the *Safe Path* of the codes. The safe path are the instructions executed in tasks from the time the tasks become non-speculative until the tasks commit. We define *Similarity* as $\frac{|I_{crit} \cap I_{safe}|}{|I_{crit} \cup I_{safe}|}$, where I_{crit} and I_{safe} are the dynamic instructions in the critical and safe paths, respectively. On average, the similarity is 15% for Olden, 21% for SPECfp, and 46% for SPECint. The fact that these numbers are low means that the critical path is typically in the speculative tasks, not in the non-speculative one. This is more the case in the Olden applications. The reason is that they have fewer squashes and higher parallelism than the other applications.

6.2 Performance and Power Impact

We now consider the performance and power impact of scheduling for task criticality in SM. Due to space limitations, we provide only a brief analysis. Figure 4(a) shows the execution time in our six environments normalized to *4H-Base*. Overall, the execution times of the *4H* environments tend to be similar. The execution times of the *1H* en-

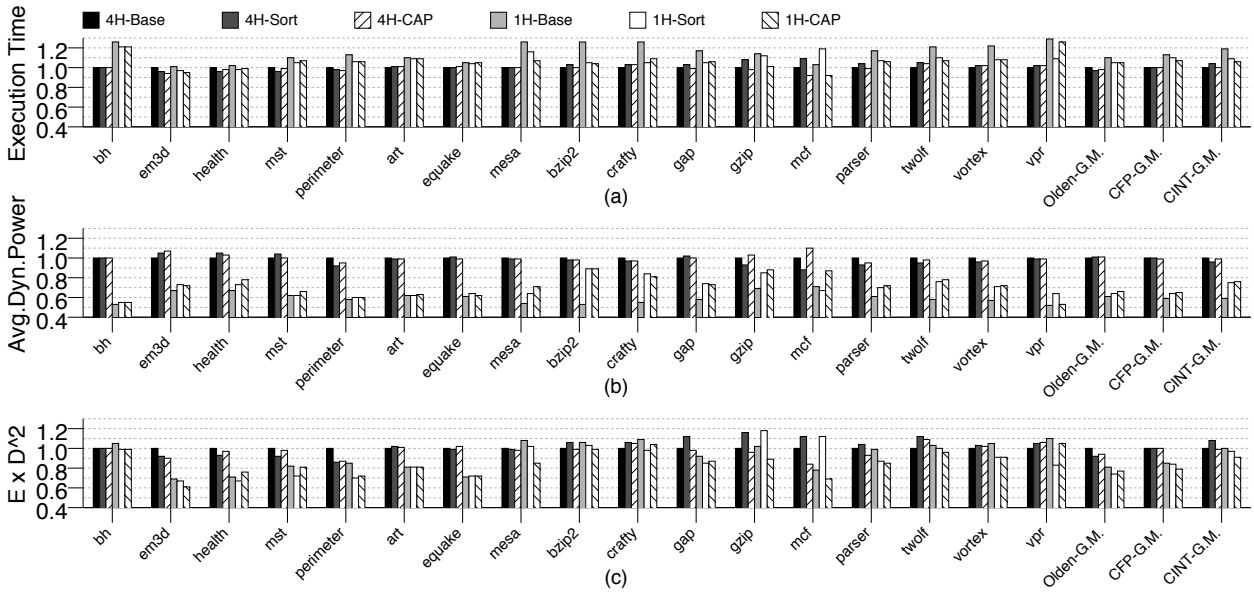

Figure 4. Execution time (a), average dynamic power consumption (b), and $E \times D^2$ (c) for different environments. All bars are normalized to *4H-Base*.

vironments are longer, but show a reduction as we go from *1H-Base* to *1H-Sort* and to *1H-CAP*. On average, thanks to criticality scheduling, the average execution time of applications on *1H-CAP* is 89–95% of that on *1H-Base*. Moreover, applications take on average only 5–7% longer to execute on *1H-CAP* than on *4H-Base*.

Figure 4(b) shows the average dynamic power consumed by the different environments. The figure shows that all the *1H* environments consume much less average power than the *4H* environments. This is due to the low consumption of three of the cores. On average, *1H-CAP* consumes 65-76% of the power of *4H-Base*.

Finally, Figure 4(c) shows $E \times D^2$ for the different environments. *1H-CAP* is the best environment, with an $E \times D^2$ that is, on average, 91–95% of that of *1H-Base* or 77–91% of that of *4H-Base*. We feel, therefore, that scheduling for criticality as in the CAP architecture is beneficial. The simpler *1H-Sort* also does well for the Olden applications — in fact, it does better than *1H-CAP*. However, for the most challenging applications, namely the SPECint codes, *1H-CAP* is substantially better: its average $E \times D^2$ is only 92% of that of *1H-Sort*.

7. Conclusion

To improve the power-efficiency of SM, this paper made three contributions: (i) it developed a novel task-criticality model for SM, (ii) it proposed the CAP architecture, which builds a task-criticality graph dynamically and uses it to make scheduling decisions; and (iii) it evaluated CAP.

Experiments with SPECint, SPECfp, and Olden applications showed that a CMP with SM runs more power-efficiently with CAP. Specifically, in a CMP with one fast core and three slow ones, the $E \times D^2$ with CAP is, on average, 91–95% of that without. Moreover, it is only 77–91% of the $E \times D^2$ in a CMP with four fast cores and no CAP. Overall, we argue that scheduling for criticality with the CAP architecture is beneficial.

References

[1] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *ISCA*, June 2000.
[2] J. Casmira and D. Grunwald. Dynamic instruction scheduling slack. In *Kool Chips workshop*, December 2000.
[3] B. Fields, R. Bodik, and M. Hill. Slack: Maximizing performance under technological constraints. In *ISCA*, May 2002.
[4] B. Fields, S. Rubin, and R. Bodik. Focusing processor policies via critical-path prediction. In *ISCA*, July 2001.
[5] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *ASPLOS*, October 1998.
[6] V. Krishnan and J. Torrellas. A chip-multiprocessor architecture with speculative multithreading. *IEEE Trans. on Computers*, September 1999.
[7] T. Li, A. R. Lebeck, and D. J. Sorin. Quantifying instruction criticality for shared memory multiprocessors. In *SPAA*, June 2003.
[8] W. Liu, J. Tuck, L. Ceze, K. Strauss, J. Renau, and J. Torrellas. POSH: A TLS compiler that exploits program structure. In *PPoPP*, March 2006.
[9] A. Moshovos, S. Breach, T. Vijaykumar, and G. Sohi. Dynamic speculation and synchronization of data dependences. In *ISCA*, June 1997.
[10] R. Nagpal and A. Bhowmik. Criticality based speculation control for speculative multithreaded architectures. In *APPT*, 2005.
[11] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. SESC simulator, January 2005. http://sesc.sourceforge.net.
[12] J. Renau, K. Strauss, L. Ceze, W. Liu, S. Sarangi, J. Tuck, and J. Torrellas. Thread-level speculation on a CMP can be energy efficient. In *ICS*, 2005.
[13] J. Renau, J. Tuck, W. Liu, L. Ceze, K. Strauss, and J. Torrellas. Tasking with out-of-order spawn in TLS chip multiprocessors: microarchitecture and compilation. In *ICS*, 2005.
[14] P. Salverda and C. Zilles. A criticality analysis of clustering in superscalar processors. In *MICRO*, 2005.
[15] J. Seng, E. Tune, and D. Tullsen. Reducing power with dynamic critical path information. In *MICRO*, December 2001.
[16] G. Sohi, S. Breach, and T. Vijaykumar. Multiscalar processors. In *ISCA*, June 1995.
[17] J. Steffan, C. Colohan, A. Zhai, and T. Mowry. A scalable approach to thread-level speculation. In *ISCA*, June 2000.
[18] D. Tarjan, S. Thoziyoor, and N. Jouppi. CACTI 4.0. Technical Report HPL-2006-86, Hewlett Packard Laboratories Palo Alto, June 2006.
[19] E. Tune, D. Liang, D. Tullsen, and B. Calder. Dynamic prediction of critical path instructions. In *HPCA*, September 2001.
[20] E. Tune, D. M. Tullsen, and B. Calder. Quantifying instruction criticality. In *PACT*, September 2002.
[21] H. S. Wang, X. P. Zhu, L. S. Peh, and S. Malik. Orion: A power-performance simulator for interconnection networks. In *MICRO*, November 2002.