Speculative Thread Decomposition Through Empirical Optimization *

Troy A. Johnson Rudolf Eigenmann T. N. Vijaykumar

School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN 47907-2035
troyj@purdue.edu eigenman@purdue.edu vijay@purdue.edu

Abstract

Chip multiprocessors (CMPs), or multi-core processors, have become a common way of reducing chip complexity and power consumption while maintaining high performance. *Speculative* CMPs use hardware to enforce dependence, allowing a parallelizing compiler to generate multithreaded code without needing to prove independence. In these systems, a sequential program is decomposed into threads to be executed in parallel; dependent threads cause performance degradation, but do not affect correctness. Thread decomposition attempts to reduce the run-time overheads of data dependence, thread misprediction, and load imbalance. Because these overheads depend on the run times of the threads that are being created by the decomposition, reducing the overheads while creating the threads is a circular problem. Static compile-time decomposition handles this problem by estimating the run times of the candidate threads, but is limited by the estimates' inaccuracy. Dynamic execution-time decomposition in hardware has better run-time information, but is limited by the decomposition hardware's complexity and run-time overhead. We propose a third approach where a compiler instruments a profile run of the application to *search* through candidate threads and pick the best threads as the profile run executes. The resultant decomposition is compiled into the application so that a production run of the application has no instrumentation and does not incur any decomposition overhead. We avoid static decomposition's estimation accuracy problem by using actual profile-run execution times to pick threads, and we avoid dynamic decomposition's overhead by performing the decomposition at profile time. Because we allow candidate threads to span arbitrary sections of the application's call graph and loop nests, an exhaustive search of the decomposition space is prohibitive, even in profile runs. To address this issue, we make the key observation that the run-time overhead of a thread depends, to the first order, only on threads that overlap with the thread in execution (e.g., in a four-core CMP, a given thread can overlap with at most three preceding and three following threads). This observation implies that a given thread affects only a few other threads, allowing pruning of the space. Using a CMP simulator, we achieve an average speedup of 3.51 on four cores for five of the SPEC CFP2000 benchmarks, which compares favorably to recent static techniques. We also discuss experiments with CINT2000.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—compilers, optimization; C.1.4 [*Processor Architectures*]: Parallel Architectures

General Terms Algorithms, Performance

Keywords chip multiprocessor, decomposition, empirical search, multi-core, thread-level speculation

1. Introduction

Architectures called single-chip multiprocessors (CMPs), or multi-core processors, help reduce chip complexity and power consumption while maintaining high performance. A CMP may be used as a conventional multiprocessor to run multiple applications concurrently; however, an individual application may need to take advantage of multiple cores for high performance because each core on a CMP may be less powerful than a traditional uniprocessor. Parallelism within a single application can come from explicitly parallel sections, but it is difficult for programmers to parallelize applications manually. Although compilers are relatively successful at parallelizing numerical applications, dependences that are not statically analyzable hinder compilers. To alleviate this problem, *speculative* CMPs [15, 30, 32, 33, 38, 39] exploit the parallelism implicit in an application's sequential instruction stream. Speculative CMPs use hardware to enforce dependence, allowing a compiler to focus on improving performance without needing to prove independence. A sequential program is decomposed into threads for a speculative CMP to execute in parallel; dependent threads cause performance degradation, but do not affect correctness. The CMP uses prediction to select and execute a sequence of threads while enforcing correctness, such that the program's output is consistent with that of its sequential execution. The CMP employs data-dependence-tracking mechanisms, keeps uncertain data in speculative storage, rolls back incorrect executions, and commits data to main memory only when speculative threads succeed. Thus, a speculative CMP provides the same programming interface as a uniprocessor while supporting the safe, simultaneous execution of potentially dependent threads – referred to as thread-level speculation (TLS).

The *decomposition problem* is to partition a program into speculatively parallel threads while optimizing run time. Thread decomposition is the critical factor in determining the performance

* This material is based upon work supported in part by the National Science Foundation under Grants No. 9974976-EIA, 0103582-EIA, and 0429535-CCF. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

PPoPP'07 March 14–17, 2007, San Jose, California, USA.

of a program executed by TLS. The key factors contributing to run-time overhead in TLS are inter-thread data dependence, inter-thread control-flow misprediction, and inter-thread load imbalance. Data dependence and misprediction cause roll backs and load imbalance causes idling. The amount of run-time overhead caused by an instance of these factors depends on the thread size. The penalty due to a data-dependence violation or misprediction depends on how late in a thread's execution the violation or misprediction is detected – the longer the thread, the later the potential detection, and the larger the penalty. Similarly, the longer the threads, the larger the potential load imbalance, and the larger penalty. Because these overheads depend on the run times of the threads that are being created by the decomposition, reducing the overheads while creating the threads is a circular problem. Static compile-time decomposition handles this problem by estimating the run times of the candidate threads, but is limited by the inherent inaccuracy of the estimates and of predicting the run-time interaction (e.g., data dependence) among threads. Dynamic execution-time decomposition in hardware has better run-time information in that hardware can measure actual run times instead of relying on estimates. Nevertheless, dynamic decomposition is limited by the decomposition hardware's complexity and run-time overhead of performing decomposition during execution. It is limited also by lacking knowledge of the overall program structure, which is useful in finding good decompositions, and by the difficulty of performing in hardware the complex tradeoffs among the various overheads of TLS.

To address these limitations of the current decomposition schemes, we propose a different approach where a compiler instruments a profile run of the application to perform an *empirical search* through candidate threads and pick the best threads (i.e., least execution times) as the profile run executes. As the profile run proceeds, it uses the naturally occurring invocations of procedures and loops to try out various candidate threads (e.g., each invocation may try out a different candidate thread). The resultant decomposition is compiled into the application so that a production run of the decomposed application has no instrumentation and does not incur any decomposition overhead. Because we use actual profile-run execution times to pick the threads, we avoid static decomposition's estimation accuracy problem (barring any inaccuracies due to differences between profile inputs and production inputs). We avoid dynamic decomposition's run-time overhead by performing the decomposition at profile time, even though we evaluate tradeoffs among TLS's various overheads during the decomposition. Also, our search uses knowledge of the overall program structure.

Our approach has some similarities to loop versioning [3, 14] and procedure cloning [6]. Whereas our approach chooses among various candidate threads, those optimizations choose among various versions of loops and procedures. While those optimizations are only locally applicable to loops and procedures, our approach is more general and tries numerous levels of parallelism spanning arbitrary sections of a program's call graph and loop nests. Those optimizations compile multiple static versions into the program and choose among them at run time, but the number of versions is limited by code expansion. By contrast, our scheme examines candidate threads only during a profile run and uses only the best threads in compilation for production runs, avoiding code expansion issues. One may think that code expansion problems remain in the profile run. Fortunately, speculative CMPs provide hardware support to dynamically coalesce all the static threads of a procedure call or loop into a single dynamic thread during execution without requiring an explicit static version of that single thread (see Section 3). Consequently, the compiler needs to generate only one version of each procedure or loop. Our profile run compares that version to alternative executions, in which some calls or loops are treated as single threads, via the hardware support. This dynamic ability obviates creation of multiple static versions, avoiding code expansion issues even in our profile run. Another paper [27] creates two versions of loops – serial and parallel – in cases where the correctness of the parallel version cannot be proved at compile time. The parallel version is executed and then checked for correctness via a run-time dependence test; if the test fails, then the serial version is executed. Two key differences to our approach are: (i) [27] assumes the parallel version is always faster than the serial version and, therefore, there is no comparison between them. By contrast, our scheme has many parallel versions (candidate threads) and has to choose among them, for which we use empirical search. (ii) Because our hardware can always guarantee correctness with any parallel version, we do not need to perform any dependence tests.

Finally, because we allow candidate threads to span arbitrary sections of the application's call graph and loop nests, an exhaustive search through the decomposition space is prohibitive, even in profile runs. In addition, because our search uses the naturally occurring invocations of procedures and loops to try out candidate threads, the number of invocations in a single run may not be enough to try out all candidate threads. To address this issue, we make the key observation that the run-time overhead of a thread depends, to the first order, only on those threads that overlap with that thread in execution (e.g., in a four-core CMP, a given thread can overlap with at most three preceding and three following threads). We call this property *speculation locality* and say that two threads are *independent by separation* if there are enough other threads between them to prevent them from executing at the same time. Speculation locality implies that, when our search replaces a candidate thread with a better thread, the TLS overheads of only those threads that overlap with the candidate thread will change. Consequently, the search does not need to revisit other threads that are independent by separation of the thread it just replaced, enabling pruning of the search space.

Our main contributions are:

- We are the first to propose that decomposition for thread-level speculation (TLS) be performed by a profile-time *empirical search* that is embedded into the program.[1]
- We show how to embed search code within an application to find the most parallelism, with low profile run-time overhead and while avoiding measurement-induced error.
- We identify properties called *speculation locality* and *independence by separation* to prune the search.
- We give all TLS overheads equal priority, even those that static approaches ignore (e.g., memory latency).
- Our results show an average speedup of 3.51 on four cores for five SPEC CFP2000 benchmarks, compared to an average speedup of 2.97 obtained with a recent static technique [16]. Empirical optimization produces results better than static approaches with less analysis effort and without the drawbacks of dynamic decomposition.

In Section 2, we discuss related work, followed by an explanation of the speculative execution model in Section 3. We discuss our optimization system in Section 4 and our results in Section 5. Section 6 concludes.

2. Related Work

Static decomposition techniques face the problem of making good compile-time tradeoffs among run-time overheads. Finding optimum program partitions in general is NP-complete [29], as is find-

[1] A related idea, adaptive speculative task parallelism, was mentioned in [19], but was considered beyond the scope of that paper.

ing thread-level parallelism [12], and the relation of overheads to thread size means that traditional static scheduling algorithms [20] are not appropriate. Consequently, a compiler uses heuristics [35] or special graph-partitioning algorithms [16] to reconcile the conflicting demands of these constraints. Loops and procedure calls commonly form threads, as in this paper and others [21]. Fundamentally, all static approaches suffer from the inherent inaccuracy of predicting the run-time duration and interaction of speculative threads. Simple profiling can help [16, 21], but the compiler must ignore many run-time effects to simplify the analysis. Manual decomposition [26, 31] is common due to the above difficulties.

Dynamic decomposition techniques implemented in hardware inspect the instruction stream looking for loops [22, 34], procedure calls [1], or cache-line access patterns [28]. Then the hardware creates threads from loop iterations, executes the code following a call in parallel with the call, or takes advantage of cache behavior. Another method [5] looks ahead in the instruction stream while executing code and marks where the next thread should begin. The look-ahead searches for instruction patterns that are known from experience to be beneficial places to begin threads. Hardware techniques cannot benefit from high-level knowledge of the program's structure or make complex tradeoffs among overheads.

Adaptive compilation systems [7, 37] compare various code versions by executing them. The different versions utilize different or reordered compiler optimizations and are generally included with the program at compile-time such that a run-time choice can be made among them. In [37], the additional versions are compiled on another processor while the main processor executes the application, and are then dynamically loaded.

3. Speculative CMP Execution Model

We use the Multiscalar architecture [30] as an example to explain a speculative CMP's execution model. Our empirical search relies on source code instrumentation and a hardware counter; the specific underlying architecture does not matter, provided that it has several options for creating threads through which the search can iterate. The primary difference among the architectures mentioned in Section 1 lies in the cache protocol they use for managing speculative storage and detecting violations. Different cache protocols impact performance, but do not change the compiler's view of the execution model [4, 11, 13, 15, 32].

Thread Execution A thread dispatcher (in hardware) fetches threads from the sequential instruction stream and dispatches them to processor cores. It uses prediction to decide which thread to dispatch next. A thread's execution may be incorrect either because the prediction was wrong, resulting in a control-dependence violation, or because an inter-thread data dependence was violated. The CMP detects both types of violations and reacts by rolling back and restarting threads as necessary [10, 13]. The oldest thread in execution (w.r.t. sequential program order) is always nonspeculative, guaranteeing progress, while all younger threads are speculative. A speculative thread keeps its uncertain data in speculative storage until it becomes the nonspeculative thread and commits changes to memory. A formal execution model can be found in Section 2 of [18]. The speedup achieved by TLS is reduced by various overheads, which we describe next.

General Per-Thread Overhead Although thread dispatch is efficient, it remains a significant overhead for small threads (i.e., less than twenty cycles). Decomposing a program into large threads reduces the significance of this overhead; however, very large threads (i.e., thousands of cycles) can overflow the speculative storage because they will include more writes to distinct memory locations. An overflow completely stalls speculative execution, until the nonspeculative thread completes and allows the next thread to become nonspeculative, freeing speculative storage. Techniques exist to reduce the speculative storage required by a program [11, 18].

Data-Dependence and Control Violation Overhead True dependences that cross thread boundaries may lead to data-dependence violations and cause rollbacks, as in Figure 1. A data-dependence violation is detected at the write reference to a memory location that was read previously by a younger thread (w.r.t. sequential program order). The reader and all younger threads are rolled back, as in Figure 1. The run time of the rolled-back threads is overhead.

Only true memory dependences (read-after-write) cause violations. Anti (write-after-read) and output (write-after-write) dependences are properly handled by buffering in the speculative storage. CMP architectures can learn to synchronize dynamically any frequently-encountered memory dependences that impede parallel execution [23]; it is difficult to predict this synchronization during decomposition. Furthermore, register dependences are specified by the compiler, allowing the hardware to communicate register values from one thread to another as appropriate [2]. Register-value communication among processor cores can be a significant overhead for programs with many small threads.

Thread mispredictions cause control-dependence violations. They are detected when an older thread completes and its actual successor differs from the predicted successor. The overhead is the run time of the rolled-back, younger threads, as in Figure 1. This situation is essentially identical to detecting a data-dependence violation at the very end of a thread.

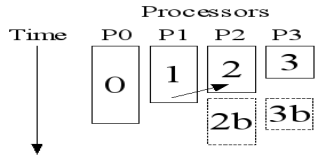

Figure 1. *Rollback due to a data or control dependence violation:* Thread numbers indicate sequential order. Thread 1 detects a violation in thread 2. Thread 2 and the younger thread 3 are rolled back, followed by the dispatch of new threads on P2 and P3. Threads 2b and 3b may or may not be the same as threads 2 and 3.

Load Imbalance Overhead Threads of unequal size can cause load imbalance, as in Figure 2. The imbalance stems from an architecture property: threads are dispatched to processor cores in a cyclic order and a core does not receive a new thread until it has committed its current thread. Because threads commit in program order, younger threads have to wait for older threads to commit. A large thread preceding (in program order) a small thread causes the small thread to wait until the large thread commits, idling execution cycles. Although maintaining a cyclic dispatch order simplifies the architecture by allowing the sequence of threads to be determined easily for rollback operations, it results in load imbalance. With more complicated hardware, it is possible to avoid this overhead by dispatching out-of-order or executing multiple threads per processor core.

While the above overheads are important for decomposition, there is one other detail that is also relevant, namely thread coalescing. The hardware supports dynamically coalescing the threads within a procedure call or loop into a single thread. Everything beneath the call or loop on the dynamic call graph (i.e., all code executed until the call's return or loop's exit) is *"collapsed"* into the thread invoking the procedure or loop. The hardware ignores thread boundaries (i.e., instructions that begin new threads) while such a call or loop executes, but still executes the other instructions. The alternative is to continue execution by beginning new threads within the called procedure or loop. It is possible for a call to the

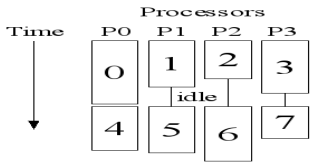

Figure 2. *Load imbalance:* Strict cyclic dispatch order in the architecture leads to load imbalance. This overhead can be avoided by supporting complex out-of-order thread dispatch or executing multiple threads per core.

same procedure to be collapsed into the calling thread for one call site, but not for another. Programs containing many calls will have small threads unless some calls are collapsed, thus this mechanism is important for exploiting coarse parallelism.

4. Profile-Time Empirical Optimization

Thread decomposition is key to reducing the above overheads. Recall from Section 1 that our approach is to instrument a profile run of the application such that it performs an *empirical search* through candidate threads and picks the best threads as it executes. The resultant decomposition is compiled into the application so that subsequent runs have no instrumentation or decomposition overhead. We examine the factors one considers when finding candidate threads and the decisions made by existing static and dynamic approaches. We examine three categories of candidate threads: loop iterations (Section 4.1.1), procedure calls (Section 4.1.2), and everywhere else (Section 4.1.3). Next, we describe the decomposition space – the variety of possible decompositions. Finally, we present the design of our optimization system and explain our implementation.

4.1 Candidate Threads

4.1.1 Loop Iterations

Loop iterations are the most obvious part of a program from which to create threads and many approaches to TLS focus on loops [8, 21, 24, 26, 31]. Loop iterations provide a run-time sequence of predictable and naturally load-balanced threads, leaving dependence as the primary overhead to potentially interfere with speculation. Nevertheless, the decision of whether to speculate is not straightforward. Due to thread dispatch overhead, it may be better to coalesce all loop iterations into a single thread (i.e., serialize the loop) if the loop is small with few iterations. Speculating on a nested loop may interrupt the coarser parallelism of an outer loop. Many loops are not perfectly nested, which makes it hard to determine at which granularity to speculate. The heuristic approach in [35] speculates on all loops because its dominant heuristic is thread prediction. The min-cut approach in [16] serializes some inner loops based on a performance estimate, but does not always make correct decisions because the compile-time estimation ignores several run-time effects (e.g, memory latency). Dynamic hardware approaches often begin threads whenever a backward branch is taken, regardless of nesting level.

4.1.2 Procedure Calls

Besides loop iterations, the other way to create large threads is to coalesce all threads in the dynamic call graph beneath a procedure call into a single thread, as described in Section 3. In this way, many static threads are dynamically collapsed into the calling thread by ignoring thread boundaries until the call returns. The two primary benefits are executing multiple calls in parallel and preventing short calls from introducing load imbalance into a sequence of larger threads. The danger is that there is parallelism within the called procedure. By collapsing the call, that parallelism is discarded with the expectation that coarser parallelism will make up for the loss; however, this is not always the case. A collapsed call may introduce a large thread into a sequence of smaller threads, causing load imbalance. Larger threads mean more work is rolled back upon a violation and make speculative buffer overflow more likely. Executing multiple calls in parallel is risky because dependences through global data are often difficult to detect. Therefore, there must be a high degree of confidence that collapsing a call will improve performance. The heuristic approach in [35] collapses only very small calls and library calls. The min-cut approach in [16] collapses some larger calls based on a performance estimate, but because the estimation views collapsing a call primarily as a way of reducing load imbalance, the vast majority of the collapsed calls are small and located in loops. Dynamic hardware approaches may execute a call in parallel with the code following the call. If the call is larger than a threshold, then the hardware breaks it into additional threads.

4.1.3 Elsewhere

There exist some places besides loops and procedure calls where it may be beneficial to begin threads. Such places are normally at points of control-flow convergence (post-dominators), or at the ends of dependence chains within lengthy calculations, and are found by the heuristics in [35] and the min-cut algorithm in [16]. Beginning threads there exploits fine-grain speculative parallelism.

4.2 Decomposition Space

As a default, the compiler assumes that it is good to speculate on all loops and calls within a program. Therefore, our search through candidate threads becomes, for each loop l of a program, to decide whether or not serializing l improves performance (similar to [36]) and, additionally, for each call site c of a program, to decide whether or not collapsing c improves performance. The search does not begin threads elsewhere. Our goal is to exploit loop-level and call-level speculation as aggressively as possible to achieve coarse-grain parallelism. Whereas smaller threads sometimes made sense in our previous work that focused on fine-grain speculation [16, 35], here they would cause significant load imbalance.

For each loop and procedure call of a program, our architecture provides three threading options, shown in Figure 3, that we label 0, 1, and 2. Option 0 (fine-grain parallel) executes as a set of threads, Option 1 (serial) executes as a single thread that is coalesced with the preceding and following threads, and Option 2 (coarse-grain parallel) executes as a single thread that is coalesced with the preceding thread only, beginning a new thread once the call returns or the loop exits. If the set of call sites is C and the set of loops is L, then there are $3^{|C|+|L|}$ possible program decompositions. Evaluating every solution is not feasible due to exponential complexity, so finding a good solution requires an intelligent search of the space. Note that other speculative architectures may provide different threading options, or more, or fewer; the technique presented in this paper can be adapted to them by varying the specific search algorithm, while leaving the empirical search mechanism intact.

In the following discussion, we use an extended call graph that also includes loops. The vertex set V is the procedures and loops of a program, while the edge set E is the calls, where we treat beginning a loop as a call. If one procedure calls another multiple times, then there are multiple edges to the callee, such that the entire graph has $|E| = |C| + |L|$ edges. A program decomposition can be represented on the graph by writing a base-3 numeric string of length one next to each edge, where each digit indicates one of the three threading options described above in Figure 3. We refer to the concatenation of strings of all edges leaving a particular vertex v as the decomposition for vertex v (i.e., for the call or loop represented

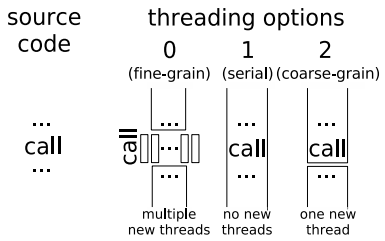

Figure 3. *Options for threading a call:* There are many threads using Option 0, one thread using Option 1, and two threads using Option 2.

by v). A complete decomposition for all $v \in V$ has $|E|$ base-3 digits, capable of representing the $3^{|E|}$ possible solutions. We make two crucial observations that are the basis of our strategy for pruning this exponential space.

OBSERVATION 1. *In a speculative CMP capable of executing N threads simultaneously, the violations described in Section 3 can occur only between two threads that have fewer than $N-1$ threads separating them in program order. If $N-1$ or more threads separate them, then the threads cannot execute simultaneously, and therefore cannot cause a violation. We say that two threads are* independent by separation *if there are enough other threads between them to prevent their simultaneous execution.*

OBSERVATION 2. *It follows that the speculation overheads of threads within a call or loop are most influenced by the $N-1$ threads immediately preceding the call or first iteration, as well as the threads within the call or loop. Additionally, the threads within a call or loop most influence the $N-1$ threads immediately following the return or last iteration. We call this property* speculation locality, *shown in Figure 4.*

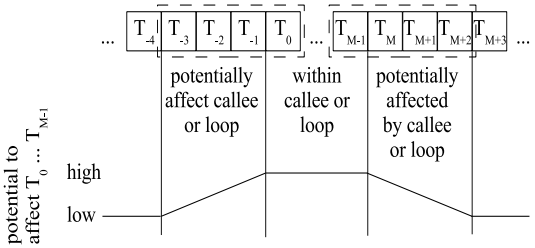

Figure 4. *Principle of Speculation Locality:* The influence of threads before and after a call or loop decreases beyond the N threads in the execution window. Here, the loop or call contains M threads and the window is $N = 4$. Typically $M > N$.

STRATEGY 1. *For each vertex v of the extended call graph, if a decomposition is first found for all of v's children before v's decomposition is determined, then it is not necessary to revisit the decomposition of v's children after determining v's decomposition.*

Strategy 1 is supported by induction on the graph. *Base Case:* A trivial decomposition can be found for leaves of the graph because they make no procedure calls and do not contain loops; i.e., their solution string is a null string. *Inductive Case:* For a given vertex v of the graph, we have, by hypothesis, a decomposition for each child. Therefore, any edge e marked with Option 0 that leaves v leads to a sequence of threads, s, that was deemed the best known decomposition for that child; i.e., Option 1 or 2 may be better, but s is the best under Option 0. Because of speculation locality, how we mark other edges besides e that leave v has *little to no effect* on s. If e is instead marked with Option 1 or 2, then s has *no effect* on the decomposition of v because it is collapsed and its thread boundaries are ignored at run time, as in Figure 5. Therefore, if we search for a decomposition for v without revisiting the decomposition of any of its children, it will not have a significant negative impact on our final solution. We continue in this manner until we find the decomposition for the root of the graph (i.e., the program's `main` procedure), and then concatenate the decomposition of each vertex to form an overall solution string.

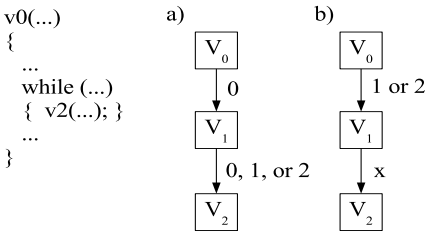

Figure 5. *Collapse and Serialization:* Each extended call graph corresponds to the code on the left: v_0 represents procedure v0, v_1 represents the while loop, and v_2 represents procedure v2. The $v_0 \rightarrow v_1$ edge represents entering the while loop and the $v_1 \rightarrow v_2$ edge represents calling v2 from the while loop. There are two edges but only five distinct base-3 solution strings ("00", "01", "02", "1x", "2x"), shown as figures (a) and (b), instead of nine (3^2). In (a), the lower digit's value matters. In (b), the lower digit's value does not matter and is marked with an "x" because v_1 is collapsed into v_0.

Due to the NP-complete nature of the problem and the use of a locality property, there is no proof of optimality for Strategy 1, but it suggests that a bottom-up[2] approach yields a good solution. A consequence of Strategy 1 is that instead of having to explore $3^{|E|}$ possible solutions, the search space is pruned to an average of $|V|3^{\beta_{avg}}$ solutions, where β_{avg} is the average branching factor of the graph (i.e., assuming on average that each vertex has β_{avg} children). Furthermore, the bottom-up approach allows solutions to be found for different procedures and loops concurrently. We now discuss the design of our optimization system, including the specific search methods it employs to further prune the decomposition space.

4.3 Design of the Optimization System

The original application that we seek to optimize along with our instrumentation code constitute the optimization system. It compares candidate threads by measuring the performance of vertices. Recall from Section 1 that we use the natural invocation order of vertices and the system has no control over this ordering. We explained our search strategy above and we describe the details of the search in two parts, the overall search and the per-vertex search. The overall search, discussed in Section 4.3.1, is coordinated separately from the searches for decompositions of particular vertices and is responsible for following the bottom-up approach of Strategy 1. Per-vertex searches (i.e., within a particular loop or procedure) are discussed in Section 4.3.2 and may proceed concurrently using one of several search methods.

[2] Recursive calls are discussed below.

4.3.1 Coordinating Overall Search

The system coordinates the bottom-up search by maintaining the state machine in Figure 6 for each vertex. There are three possible states for a vertex v:

- `Not Ready` ($\mathcal{NR}$) – It is not yet time to determine the decomposition at v. At least one of v's non-recursive children is not `Done`. $\mathcal{NR}$ is the initial state of all non-leaf vertices.
- `Ready` ($\mathcal{R}$) – A decomposition is currently being sought for v. All of v's non-recursive children are `Done`.
- `Done` ($\mathcal{D}$) – A decomposition has been found for v, or v is a leaf.

We do not want the system to wait on vertices that the program never actually executes, such as error routines. Fortunately, for the benchmarks we examine, nearly all such procedures are leaf calls, and we optimize by putting leaves directly in the `Done` state so their callers do not need to wait. Note that if a recursive call is collapsed, it is always collapsed at the top level; collapsing further down the recursion tree requires cloning the entire procedure, which we wish to avoid.

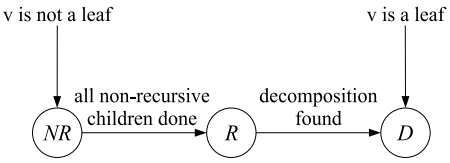

Figure 6. *State transitions: this figure shows the rules used to determine when to change vertex state. We optimize by placing leaf nodes directly into the* `Done` *state.*

4.3.2 Search Per Extended-Call-Graph Vertex

Recall that the decomposition for a particular vertex v is the set of base-3 solution strings of all edges leaving v. The major concern is how many potential solutions need to be examined before settling on a final solution. Fewer is better because if the search needs to examine X solutions and the program invokes v only Y times, where $X > Y$, then it needs multiple runs of the program to find the best solution. We wish to avoid this situation whenever possible because the state of the search would need to be saved after each run and loaded before the next run. Another reason for minimizing the number of examined solutions is to provide some leeway such that it can skip the first measurement (when caches will suffer cold misses) and then obtain multiple samples for an average. Note that it is usually unnecessary to find decompositions for the topmost vertices; for example, consider that the `main` procedure is called only once, but it is so high on the graph that any non-zero values in its solution string would serialize large chunks of the application and yield near-sequential performance.

We primarily discuss the following search per-vertex methods: exhaustive, linearly-independent, control-flow, greedy, and hierarchical. We implement and evaluate only the greedy, hierarchical, and (for very small cases) exhaustive methods because they are most promising. We describe the others for completeness.

Exhaustive Search Exhaustive search requires checking 3^{β_v} solutions, where β_v is the branching factor of v. This approach is useful when $\beta_v \leq 3$, but larger values will slow our search significantly. Vertices lower on the graph must complete before moving on to higher vertices, so whereas a $\Theta(3^n)$ algorithm might normally be acceptable for some small values larger than three, in this case it is not acceptable because it impedes the overall search.

Linearly-Independent Search At the other extreme, a linearly-independent search assumes that none of the digits in the solution have anything to do with each other, and checks only β_v solutions. For example, if the string $s_1 = 0100$ improves performance and the string $s_2 = 0010$ improves performance, then the combination $s_3 = s_1 \vee s_2 = 0110$ is assumed to improve performance. We have observed that this assumption is not reasonable; such a combination may decrease performance or show superlinear improvement. A performance decrease occurs when the combination creates a long sequence of ones that unnecessarily serializes a large amount of code. A superlinear improvement occurs when two neighboring calls are brought into the same thread and the calls have data-dependences between them; this effect can be surprising because the dependence can be on a global variable accessed by distant grandchildren of the calls, which is not discernible by looking at the source code for v alone.

Exploiting Control-Flow Because exhaustive search and linear-independence will not work in general, we next look for ways to divide the search into subproblems. One way is to exploit v's control-flow structure in conjunction with independence by separation. Consider the example in Figure 7a, where there are two mutually exclusive paths within v. The extended call graph is shown in Figure 7b. There are two calls to $v1$, so the $v \rightarrow v_1$ edge is duplicated. The branching factor, β_v, is 6, so the decomposition of v has $3^6 = 729$ possible solutions. The control flow of v is shown in Figure 7c. Notice that the partial solutions for the left and right paths are independent. Also, if an interior call (v_{1a}, v_2, v_3) of the control flow graph, shown in Figure 7c, is not collapsed, then it typically represents many threads – possibly hundreds. Therefore, it is reasonable to apply independence by separation across non-collapsed calls, treating the partial solutions on either side as independent. Instead of checking $3^6 = 729$ possible solutions, it is necessary to check 183, counting as in Table 1, or about 25% of the original number.

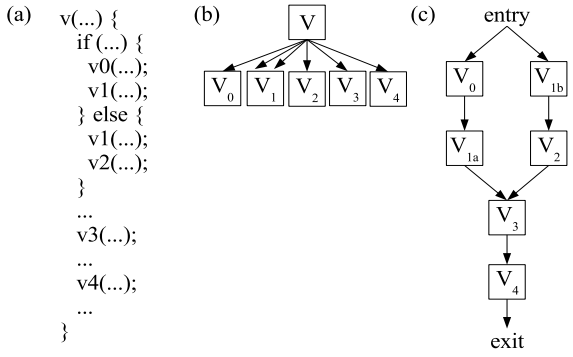

Figure 7. *An opportunity for pruning the search using both independent paths and independence by separation.*

Table 1. *Counting Decompositions:* The first column identifies the case under consideration, the second column shows how we count the solutions, and the third column shows the simplified numbers.

$v_3 = 0$	$3^2 + 3^2 + 3$	21
$v_3 \neq 0$, $v_{1a} = 0$, $v_2 = 0$	$3 + 3 + 2 * 3$	12
$v_3 \neq 0$, $v_{1a} \neq 0$, $v_2 = 0$	$3^2 * 2^2 + 3$	39
$v_3 \neq 0$, $v_{1a} = 0$, $v_2 \neq 0$	$3 + 3^2 * 2^2$	39
$v_3 \neq 0$, $v_{1a} \neq 0$, $v_2 \neq 0$	$(3 * 2 + 3 * 2) * 3 * 2$	72
total		183

Greedy-Local Search (Hill Climbing) Although exploiting control-flow is useful, there are two problems: (i) reducing the number of candidate solutions by $\frac{1}{2}$ or $\frac{1}{4}$ is not enough to make the exponential space manageable, and (ii) many procedures contain long sequences of calls and loops that are executed unconditionally and therefore lack any control flow to exploit. Local search algorithms, which operate by maintaining a current solution and moving to neighboring solutions, can significantly reduce the number of solutions considered. The most basic of these algorithms is greedy-local search, sometimes called hill-climbing search, which always moves to the best neighbor. To apply this technique, the search begins with a solution string containing only zeros (i.e., speculate on everything). The last digit is then varied, such that "...001" and "...002" are considered. Whichever of these three solutions has the least run time provides the last digit for the final solution. For example, if "...002" was best, then the search moves on to considering "...012" and "...022", and so forth until all the digits have been selected. The search could begin with the first digit, but either way it is necessary to examine $2\beta_v + 1$ solutions for a β_v-digit string. This algorithm can be surprisingly effective. For example, by applying greedy-local search to a particular procedure from SPEC CPU2000 that has $\beta_v = 8$ and a simulated, sequential run time of 82,424 cycles, we found a decomposition that took 21,564 cycles after examining $2 * 8 + 1 = 17$ of $3^8 = 6,561$ potential solutions. The ideal solution (as estimated by manual decomposition) takes 21,377 cycles, so we came within 1% of the ideal solution after considering less than 0.3% of all solutions.

Two-Stage Hierarchical Search We wish to improve upon the greedy method in an inexpensive way. The most significant limitation of the method is that it is short-sighted; by starting with the finest-grain decomposition, it can reach a coarser granularity, but not the coarsest. To see this effect, consider Figure 8a and suppose that dependences are such that the calls can run in parallel without significant speculation overhead. An obvious solution to try is "22...22", shown in Figure 8b, which places each call in a separate thread. The greedy method, however, will rarely reach this string on its final attempt because of load imbalance in the intermediate steps. For example, the intermediate step "00...22..." likely suffers some imbalance at the 0-2 transition, causing the greedy approach to toss a few ones into the string with mixed results. Similarly, the greedy method will often ignore coarser parallelism from grouping the calls, as in Figure 8c. Therefore, we perform a two-stage search, with the initial solution as all 2s. The first stage ("merge") is a mesh search that scans the string β_v times while changing a single 2 to a 1. The state ends early if nothing is changed during a scan. At best, nothing changes during the first scan and it tries β_v additional solutions. At worst, it introduces a single 1 each pass for an additional $\sum_{k=1}^{\beta_v} k = \frac{1}{2}\beta_v(\beta_v + 1)$ solutions. The second stage ("divide") is a greedy, single-scan search that tries a 0 at each position, for an additional β_v attempted solutions. By contrast to greedy search, which was $\Theta(2\beta_v + 1)$, hierarchical search is $\Omega(2\beta_v + 1)$ and $O(\frac{1}{2}\beta_v^2 + \frac{3}{2}\beta_v + 1)$.

Other Methods There are many possible search methods besides those above. For example, we considered trying simulated annealing and genetic selection, but both of them required trying an unacceptably large number of solutions. Complex per-vertex search methods impede the overall, bottom-up search, so we focus on methods that are simple and effective.

4.4 Implementation

We use source transformations, described in Section 4.4.1, to convert a program into the form that is most easily interfaced with our instrumentation. Then we add additional code, described in Section 4.4.2, to measure and record the performance of procedures.

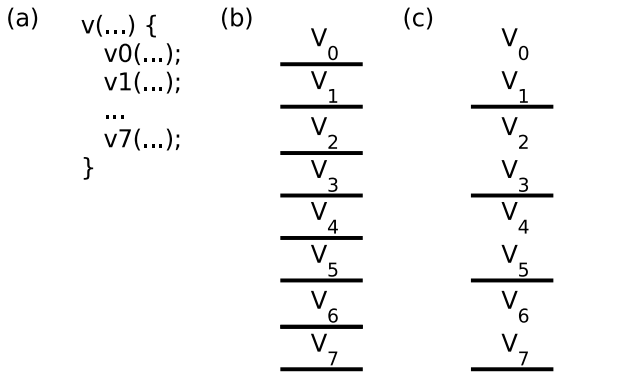

Figure 8. *Two-stage merge & divide effect:* (a) shows the original code. (b) shows thread boundaries for a solution string of "22222222." (c) shows thread boundaries for a solution string of "12121212."

We perform both the transformations and the instrumentation using the Cetus compiler infrastructure [17]. Section 4.4.3 explains how decompositions are varied at run time, and Section 4.4.4 explains how measurement error is avoided.

4.4.1 Preliminary Transformations

Loops to Procedures We apply our technique by converting loop bodies into procedures. When combined with the architecture's ability to coalesce threads (see Section 3), this transformation enables trying many levels of parallelism without experiencing code explosion from loop versioning. "Microtasking" loops is common in other parallel programming models.

Single Return There is code that must be executed just before a procedure returns. For procedures with multiple return statements, it is inconvenient to add that code in multiple places. Therefore, a procedure is transformed such that it has only a single return statement. The final return statement is modified to return the value of a new local variable; all other return statements are modified to store a value to that local variable and then jump to the final return.

Nested Call Elimination The compiler needs to place code around each procedure call, so calls that have the results of other calls passed as their arguments pose a problem: placing code around such a call would encompass more than one call. Therefore, it separate the calls using explicit temporary variables.

4.4.2 Measuring Performance

To measure the performance of a decomposition, the compiler inserts code at the beginning and end of each procedure. For example, given procedure `procV`:

```
procV(...) {
  provV0(...);
  provV1(...);
}
```

The compiler converts it to the following code:

```
procV() {
    static int _need_record = 1;
    static unsigned char* _vector = NULL;
    unsigned long long _begin;

    /* one-time initialization code */
    if (_vector == NULL)
    {
      /* allocate a β_v-length zero-filled array */
      _vector = tls_init_func(''procV'', procV,
        2, &_need_record);
```

```
    /* setup up call-graph dependencies */
    tls_callgraph(procV, procV0);
    tls_callgraph(provV, procV1);

    /* set to NR or R depending on children */
    tls_check_state(procV);
  }

  if (_need_record)
    _begin = simtime();

  /* begin transformed procedure's code */
  TLS_DECIDE(0, procV0(...));
  TLS_DECIDE(1, procV1(...));
  /* end transformed procedure's code */

  if (_need_record)
    tls_record(procV, _begin);
}
```

Our initialization code executes only once, during the invocation whose timing we discard anyway to avoid cold cache misses and a potential timing error if `procV` transitions to the $\mathcal{R}$ state while it is executing. The `_vector` array corresponds directly to the solution string for `procV` and contains only the values 0, 1, and 2. While `procV` is in state $\mathcal{R}$, `_vector` contains the solution string that is currently being measured. When `procV` is in state $\mathcal{D}$, `_vector` contains the best solution found. The `_need_record` variable is used to shut off the optimization mechanism once `procV` has been optimized. While it is turned on, the `simtime` call is used to determine the number of cycles that have elapsed since the beginning of the program; the second `simtime` call is hidden within `tls_record`. The simulator for the CMP implements `simtime` and returns a well-defined value since all of the cores execute in lockstep.

4.4.3 Changing Decompositions at Run Time

Call-site versioning is used to vary the decompositions. Recall from Section 3 that whether a procedure executes in serial or in parallel is a property of the call site; *there is no need to create multiple source-code versions of the called procedures themselves.* An `if` statement is placed around each call site to determine whether the call should execute in serial or in parallel. The TLS_DECIDE macro used above is defined as follows:

```
#define TLS_DECIDE(var, call) \
 ((_vector[var] == 0) ? call : \
 ((_vector[var] == 1) ? call : \
  call))
```

The decision is made by indexing `_vector` with the lexical order of the call in the procedure (e.g., for the first call, an index of zero is used). The code forces three versions of the call into the application's binary; the compiler's back end looks for these triple calls, marks the first with speculation option 0, the second with speculation option 1, and the third with speculation option 2. (Alternative approaches are self-modifying code, or expanding the instruction set to include a conditional-speculation opcode, but those were more difficult to add to our existing infrastructure.) The logic that updates `_vector` is located in `tls_record`. Whenever enough samples for a string have been recorded, `tls_record` advances the string to the next string to be measured. The next string depends on which search method from Section 4.3.2 is active. When `tls_record` detects that the final string (the identity of which depends on the search method) has been measured, it sets the string to the best string, moves the procedure to state $\mathcal{D}$, moves callers as necessary to state $\mathcal{R}$, and sets `_need_record` to zero to disable future measurements.

4.4.4 Avoiding Measurement-Induced Error

When timing a region of code executing on a speculative CMP, code outside the timed region can affect the measurement. This behavior is contrary to what happens if the code is executed on a sequential processor because code following the second `simtime` call (within `tls_record`) may cause a violation if there is no thread boundary in between. Another problem, due to load imbalance, is that the second `simtime` call may occur before all of the code in the timed region has finished. To prevent these effects, `tls_record` uses the following code to wait until it is the nonspeculative ("head") thread to make the measurement. The `simishead` call checks if it is executed by the head thread:

```
do {
  end_time = simtime();
} while (!simishead());

cycles = end_time - begin_time;
```

5. Performance Evaluation on SPEC CPU2000

We evaluate our technique in detail using benchmarks from SPEC CFP2000 and discuss similar experiments with SPEC CINT2000. The programs are instrumented with the Cetus [17] compiler and compiled with the Multiscalar GCC compiler at optimization level O2, which supports C programs and Fortran 77 programs via `f2c` [9]. Results for some CFP2000 benchmarks from that subset are not included due to current limitations of Cetus [17] (i.e., we use the intersection of CFP2000 currently supported by both compilers). Our search finds the decomposition by using the *train* data set for the profile run's input, but the final evaluation uses the *ref* data set. Table 2 shows the parameters we use for the Multiscalar simulator. Table 3 shows the serial instructions-per-cycle (IPC) for each benchmark and the relative performance of the greedy and hierarchical methods, taken as a *cycle ratio*. Both the greedy and the hierarchical method use exhaustive search when $\beta_v \leq 3$. We compare to the conservative approach from [35] and a more advanced static approach from [16] that used dependence-arc and branch profiles. The greedy method yields the best improvement on CFP2000. Both search methods, on average, improve over the static methods.

Table 2. Simulator Configuration

CPUs	4 dual-issue, out-of-order
L1 i-cache	64KB, 2-way, 2-cycle hit
L1 d-cache	64KB, 2-way, 3-cycle hit, 32-byte block, byte-level disambiguation
Rollback Buffer	64 entries
Reorder Buffer	32 entries
Load/Store Queue	32 entries
Function Units	2 Int, 2 FP, 2 Mem
Branch Predictor	path-based, 2 targets
Thread Predictor	path-based, 4 targets
Descriptor Cache	16KB, 2-way, 1-cycle hit
Shared L2	2MB, 8-way, 64-byte block, 12-cycle hit and transfer
L1/L2 Connect	Snoopy split-transaction bus, 128-bit wide
Core to Core Latency	10 cycles
Memory Latency	300 cycles

Figure 9 shows the cause of our speedup in terms of how the speculation overheads are reduced. Much of the speedup comes from reduced load imbalance, as in `applu`, `art`, and `mgrid`. The

Table 3. *CFP2000 Baseline vs Improved Performance:* These results were obtained using cycle-accurate simulations of representative program regions (e.g., skip data initialization and then time one billion instructions) with the *ref* data set as input.

SPEC CFP 2000	Single Thread IPC	[35]'s Speedup	[16]'s Speedup	Greedy Speedup	Hier. Speedup
applu	0.48	1.98	2.11	3.37	3.21
art	0.19	2.06	2.43	3.00	3.00
equake	0.56	1.49	1.79	1.92	1.70
mgrid	0.35	5.41	5.54	6.09	6.09
swim	0.17	4.51	4.51	4.51	4.51
g. mean		2.72	2.97	3.51	3.39

`swim` benchmark is simple enough that all methods find a good solution. Memory latency improves for `applu`, but remains mostly the same for the others. The overhead tradeoff is most evident for `equake` in which load imbalance and memory latency get worse, but dependence, misprediction, and dispatch overhead improves.

Table 4 shows that we improve the average thread size of the applications. A larger average thread size helps amortize speculation overheads and provides a coarser level of parallelism, which is generally good. When increasing average thread size manually or using static compiler analysis, it is easy to accidentally reduce performance by serializing too much of the code or by serializing the wrong parts; empirical optimization provides a safer way to reap the benefits of large thread size.

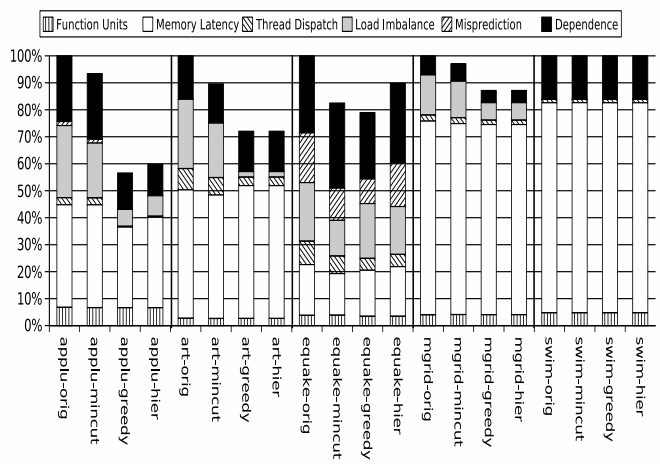

Figure 9. SPEC CFP2000 relative impact of reduced overheads compared to 100% of [35]'s overhead: orig is [35], mincut is [16], and greedy and hier are our two search methods.

Table 4. Thread Size: We show the average instructions per thread for each of the benchmarks, and then provide an overall average.

SPEC CFP 2000	[35]'s Insns/ Thread	[16]'s Insns/ Thread	Greedy Insns/ Thread	Hier. Insns/ Thread
applu	36.7	46.2	296.0	223.3
art	15.7	19.3	27.4	27.4
equake	15.5	19.6	27.2	26.8
mgrid	113.9	115.4	160.6	160.6
swim	101.9	112.8	101.9	101.9
mean	56.7	62.7	122.6	108.0

To demonstrate that our transformations and instrumentation do not substantially disturb the profile of the benchmarks, we apply all transformations and insert the instrumentation, but run the benchmarks natively. Special `simtime` and `simishead` calls are linked with the benchmarks, such that the first returns a random value and the second returns true. Therefore, the decisions made by the system have no effect and we can compare to the run time of the unmodified benchmarks. The results are shown in Table 5.

Table 5. Native Run Time (1.6 GHz AMD Athlon, GCC 3.2 -O2, Linux 2.4, *train* Data Set, Mean of Five Runs) and Beta Values: Times were collected using the `time(1)` program with output redirected to files. Run times using the hierarchical search method are comparable and not shown. Slowdown is Orig. divided by Greedy. `art` is unusual in that the transformations caused speedup; we verified that both versions of each benchmark produced exactly the same output using `diff(1)`.

SPEC CFP2K	Orig. (s)	Greedy (s)	Slowdown	β_{avg}	β_{max}
applu	24.02	38.20	0.63	1.98	15
art	76.45	46.45	1.65	2.42	30
equake	56.92	61.36	0.93	2.25	16
mgrid	28.38	39.89	0.71	1.93	7
swim	21.41	22.34	0.96	2.07	7

The same experiments with CINT2000 show that the hierarchical search generally performs better than greedy, but also that *sampling error* is a problem and leads to inconsistent performance improvement and degradation as compared to static decomposition, shown in Table 6. Although for each candidate vertex decomposition the average of several invocations is taken, the search algorithm may be varying a digit in the solution string that corresponds to a call or loop that is not executed during every invocation. Waiting for invocations that execute the code of interest is not feasible because it impedes the search and requires too many profile runs. This sampling error is not significant in the numerical, CFP2000 benchmarks because their procedures contain fewer branches. Basic solutions to this problem are known [25], but would involve more expensive instrumentation that may disturb the profiling behavior to an extent that renders the measurements meaningless for predicting how the original, unmodified program will perform. Making that instrumentation feasible is beyond the scope of this paper.

Table 6. *CINT2000 Baseline vs Improved Performance:* These results were obtained using cycle-accurate simulations of representative program regions (e.g., skip data initialization and then time one billion instructions) with the *ref* data set as input.

SPEC CINT 2000	Single Thread IPC	[35]'s Speedup	[16]'s Speedup	Greedy Speedup	Hier. Speedup
bzip2	0.70	1.01	1.09	1.07	1.17
gzip	0.72	1.27	1.35	1.11	1.17
mcf	0.07	1.01	1.63	1.07	1.09
parser	0.51	0.87	1.24	1.20	1.18
vpr	0.63	1.38	1.09	1.38	1.38
g. mean		1.09	1.27	1.16	1.19

6. Conclusions

We presented the technique of using empirical search at profile time to decompose a program into threads for execution on a speculative CMP. The new method compares favorably against previous static methods, improving speedup on four cores from 2.97 to 3.51 for five benchmarks from SPEC CFP2000. Our technique can be applied to various speculative architectures because it is independent of specific architecture parameters.

References

[1] H. Akkary. *Dynamic MultiThreaded Processor*. PhD thesis, Portland State University, June 1998.

[2] S. E. Breach, T. N. Vijaykumar, and G. S. Sohi. The Anatomy of the Register File in a Multiscalar Processor. In *Proceedings of the 27th International Symposium on Microarchitecture*, pages 181–190, November 1994.

[3] M. Byler et al. Multiple Version Loops. In *Proceedings of the International Conference on Parallel Processing*, pages 312–318, August 1987.

[4] M. Cintra, J. F. Martínez, and J. Torrellas. Architectural Support for Scalable Speculative Parallelization in Shared-Memory Multiprocessors. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 13–24, June 2000.

[5] L. Codrescu and D. S. Wills. On Dynamic Speculative Thread Partitioning and the MEM-Slicing Algorithm. *Journal of Universal Computer Science*, 6(10):908–914, 2000.

[6] K. Cooper, M. Hall, and K. Kennedy. A Methodology for Procedure Cloning. *Computer Languages*, 19(2):105–117, April 1993.

[7] P. Diniz and M. Rinard. Dynamic Feedback: An Effective Technique for Adaptive Computing. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 71–84, May 1997.

[8] Z.-H. Du et al. A Cost-Driven Compilation Framework for Speculative Parallelization of Sequential Programs. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 71–81, June 2004.

[9] S. I. Feldman, D. M. Gay, M. W. Maimone, and N. L. Schryer. A Fortran to C Converter. Technical report, AT&T Bell Laboratories, March 1995.

[10] M. Franklin and G. S. Sohi. ARB: A Hardware Mechanism for Dynamic Reordering of Memory References. *IEEE Transactions on Computers*, pages 552–571, May 1996.

[11] M. J. Garzarán et al. Tradeoffs in Buffering Memory State for Thread-Level Speculation in Multiprocessors. In *Proc. of the 9th IEEE Symp. on High-Performance Computer Architecture*, February 2003.

[12] M. Girkar and C. D. Polychronopoulos. Extracting Task-Level Parallelism. *ACM Transactions on Programming Languages and Systems*, 17(4):600–634, July 1995.

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

[14] R. Gupta and R. Bodik. Adaptive Loop Transformations for Scientific Programs. In *IEEE Symposium on Parallel and Distributed Processing*, pages 368–375, October 1995.

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

[16] T. A. Johnson, R. Eigenmann, and T. N. Vijaykumar. Min-Cut Program Decomposition for Thread-Level Speculation. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 59–70, June 2004.

[17] T. A. Johnson et al. Experiences in Using Cetus for Source-to-Source Transformations. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, pages 1–14, September 2004.

[18] S. W. Kim et al. Reference Idempotency Analysis: A Framework for Optimizing Speculative Execution. In *Proc. of the Symposium on Principles and Practice of Parallel Programming*, pages 2–11, 2001.

[19] B. Kreaseck, D. Tullsen, and B. Calder. Limits of Task-Based Parallelism in Irregular Applications. In *Proc. of the International Symposium on High Performance Computing*, October 2000.

[20] Y.-K. Kwok and I. Ahmad. Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. *ACM Computing Surveys*, 31(4):406–471, 1999.

[21] W. Liu et al. POSH: A TLS Compiler that Exploits Program Structure. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, March 2006.

[22] P. Marcuello, A. Gonzalez, and J. Tubella. Speculative Multithreaded Processors. In *Proceedings of the International Conference on Supercomputing*, 1998.

[23] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi. Dynamic Speculation and Synchronization of Data Dependences. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 181–193, June 1997.

[24] K. Olukotun, L. Hammond, and M. Willey. Improving the Performance of Speculatively Parallel Applications on the Hydra CMP. In *Proceedings of the International Conference on Supercomputing*, pages 21–30, June 1999.

[25] Z. Pan and R. Eigenmann. Rating Compiler Optimizations for Automatic Performance Tuning. In *Proceedings of Supercomputing (SC)*, November 2004.

[26] M. K. Prabhu and K. Olukotun. Exposing Speculative Thread Parallelism in SPEC2000. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, June 2005.

[27] L. Rauchwerger and D. Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 218–232, June 1995.

[28] E. Rotenberg et al. Trace Processors. In *Proceedings of the 30th International Symposium on Microarchitecture*, December 1997.

[29] V. Sarkar and J. Hennessy. Partitioning Parallel Programs for Macro-Dataflow. In *Proceedings of the Conference on LISP and Functional Programming*, pages 202–211, 1986.

[30] G. Sohi, S. Breach, and T. N. Vijaykumar. Multiscalar Processors. In *Proceedings of the 22nd International Symposium on Computer Architecture*, June 1995.

[31] J. G. Steffan. *Hardware Support for Thread-Level Speculation*. PhD thesis, Carnegie-Mellon University, April 2003.

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

[33] J.-Y. Tsai and P.-C. Yew. The Superthreaded Architecture: Thread Pipelining with Run-Time Data Dependence Checking and Control Speculation. In *Proc. of the International Conference on Parallel Architecture and Compiler Techniques*, pages 35–46, October 1996.

[34] J. Tubella and A. Gonzalez. Control Speculation in Multithreaded Processors through Dynamic Loop Detection. In *Proceedings of the 4th IEEE Symposium on High-Performance Computer Architecture*, February 1998.

[35] T. N. Vijaykumar and G. S. Sohi. Task Selection for a Multiscalar Processor. In *Proceedings of the 31st International Symposium on Microarchitecture*, December 1998.

[36] M. J. Voss and R. Eigenmann. Reducing Parallel Overheads Through Dynamic Serialization. In *Proceedings of the International Parallel Processing Symposium*, pages 88–92, 1999.

[37] M. J. Voss and R. Eigenmann. High-Level Adaptive Program Optimization with ADAPT. In *Proc. of the Symposium on Principles and Practice of Parallel Programming*, pages 93–102, 2001.

[38] E. Waingold et al. Baring It All to Software: Raw Machines. *IEEE Computer*, 30(9):86–93, 1997.

[39] Y. Zhang, L. Rauchwerger, and J. Torrellas. Hardware for Speculative Run-Time Parallelization in Distributed Shared-Memory Multiprocessors. In *Proceedings of the 4th IEEE Symposium on High-Performance Computer Architecture*, pages 162–173, February 1998.