Min-Cut Program Decomposition for Thread-Level Speculation *

Troy A. Johnson, Rudolf Eigenmann, T. N. Vijaykumar
{troyj, eigenman, vijay}@ecn.purdue.edu
School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN 47907

ABSTRACT

With billion-transistor chips on the horizon, single-chip multiprocessors (CMPs) are likely to become commodity components. Speculative CMPs use hardware to enforce dependence, allowing the compiler to improve performance by speculating on ambiguous dependences without absolute guarantees of independence. The compiler is responsible for decomposing a sequential program into speculatively parallel threads, while considering multiple performance overheads related to data dependence, load imbalance, and thread prediction. Although the decomposition problem lends itself to a min-cut-based approach, the overheads depend on the thread size, requiring the edge weights to be changed as the algorithm progresses. The changing weights make our approach different from graph-theoretic solutions to the general problem of task scheduling. One recent work uses a set of heuristics, each targeting a specific overhead in isolation, and gives precedence to thread prediction, without comparing the performance of the threads resulting from each heuristic. By contrast, our method uses a sequence of balanced min-cuts that give equal consideration to all the overheads, and adjusts the edge weights after every cut. This method achieves an (geometric) average speedup of 74% for floating-point programs and 23% for integer programs on a four-processor chip, improving on the 52% and 13% achieved by the previous heuristics.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*compilers, optimization*; C.1.4 [**Processor Architectures**]: Parallel Architectures; G.2.2 [**Discrete Mathematics**]: Graph Theory—*graph algorithms*

*This material is based upon work supported in part by the National Science Foundation under Grants No. 9703180, 9975275, 9986020, and 9974976. 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.

PLDI'04, June 9–11, 2004, Washington, DC, USA.

General Terms

Algorithms, Performance

Keywords

Thread-Level Speculation, Chip Multiprocessor, Min-Cut, Program Decomposition, Partitioning

1. INTRODUCTION

Single-chip multiprocessors (CMPs) are likely to become commodity components within a few years, as the number of transistors per chip crosses the one billion mark. CMPs may be operated as conventional multiprocessors, where explicit parallelism is exploited from concurrently running applications or parallel sections within a single application. It is difficult, however, for programmers to parallelize applications manually. Compilers have been successful parallelizing numerical applications; non-numerical codes confound compilers with dependences that are not statically analyzable. To alleviate this problem, *speculative CMPs* [10, 26, 29, 30, 32, 36] have been proposed to exploit the parallelism implicit in an application's sequential instruction stream. Because speculative CMPs use hardware to enforce dependence, the compiler can improve performance by speculating on ambiguous dependences without absolute guarantees of independence.

Speculative CMPs provide the compiler with the same interface as a standard, sequential processor while supporting the safe, simultaneous execution of potentially dependent threads. The compiler may view a speculative CMP as a multiprocessor in which the simultaneous execution of dependent threads results in performance degradation rather than incorrect execution, and can select threads to optimize run time. The compiler is responsible for decomposing the control-flow graph (CFG), and hence the sequential instruction stream, into these speculatively parallel threads. The compiler creates *speculative threads* by inserting boundary marks into the sequential instruction stream to tell the CMP where to speculate; that is, which code segments to try to execute in parallel with each other. The CMP uses prediction to select and execute a set of threads while enforcing correctness, such that the program's output is consistent with that of its sequential execution. To enforce correctness, the CMP employs data-dependence-tracking mechanisms, keeps uncertain data in speculative storage, rolls back incorrect executions, and commits data to the memory system only

when speculative threads succeed.

Because *thread decomposition* is a critical factor in determining the performance achieved by a speculatively-threaded program, the decomposition scheme used by the compiler is key to the success of the speculative approach. To perform thread decomposition, the compiler faces multiple performance overheads related to data dependence, load imbalance, thread size, and thread prediction. Ideally, no data dependence should cross a thread boundary to avoid dependence-synchronization delays and dependence-violation rollbacks; thread sizes should be chosen to avoid load imbalance; a thread should be large enough to amortize its dispatch overhead, but small enough such that all of its speculative data can be buffered; and thread sequences should be predictable to avoid misprediction rollbacks. Finding optimum program decompositions in general is NP-complete [25].

Because this problem requires partitioning a program, it naturally lends itself to a min-cut-based approach. Certainly, others have used graph-theoretic approaches to solve compiler problems. Most relevant are static scheduling algorithms [18] that map threads from an explicitly parallel program onto processors while minimizing interthread-communication overhead and load imbalance. These algorithms share a subset of the goals – minimize dependences and load imbalance – of this decomposition problem. Nevertheless, there is a fundamental difference between scheduling and our decomposition. In our case, *all* the overheads due to data dependence, thread-sequence misprediction, and load imbalance depend on the size of the threads. Because an edge's weight represents the run-time overhead incurred by cutting the edge, our weights depend on thread sizes and *change* as the algorithm progresses and newer threads are made. In contrast, the weights in scheduling are fixed at the beginning and do not change. Decomposition using fixed weights results in poor speculative-threaded performance because the weights lack a relationship to overhead. Apart from the changing weights, most scheduling algorithms [18] make assumptions that are not applicable to our problem, such as unlimited processors, uniform thread run time, or constant communication cost (i.e., weights are the same for all edges).

Some past approaches for decomposition have focused exclusively on loops [17, 20, 27]. Unfortunately, non-loop code sections are crucial for non-numerical programs, which are the primary target for speculative CMPs. To that end, [31] applies several different heuristics to build threads from loops and non-loops. The heuristics, however, do not consider load imbalance and use limited dependence and prediction information. The heuristics conservatively attempt to make threads out of all loop bodies, and terminate threads at all but the smallest function calls, ignoring opportunities for coarser parallelism. The heuristics target the overheads of data dependence, load imbalance, and thread prediction *separately*, and give precedence to thread prediction, without comparing the performance of the threads resulting from each isolated heuristic. In contrast to [31], this paper chooses the best-performing threads *by giving equal consideration to all the overheads* using a min-cut-based approach. Both techniques are back-end compiler algorithms that do not modify the application's source code.

We apply min-cut on the CFG of each procedure of an application. In any min-cut-based approach including ours, where overhead due to data dependence and thread prediction can be represented as edge weights, load imbalance does not lend itself to being represented as edge weight. Consequently, we use *balanced min-cut* [35]. For balancing, [35] modifies a min-cut to reduce the difference between the vertex sizes of the cut's two vertex sets. Our balancing is significantly more sophisticated in that it reduces the overall run time of the threads resulting from the cut. We employ an abstract-execution-based scheme to estimate run times of candidate thread sets. Combined with the previously-mentioned requirement that our edge weights change as newer cuts are made, our approach performs a sequence of balanced min-cuts where the edge weights are adjusted after each balanced cut.

Our main contributions are:

- We are the first to map the speculative-thread decomposition problem onto a graph-theoretic framework. By using a sequence of balanced min-cuts and adjusting the edge weights after every cut, we give equal consideration to the overheads of data dependence, thread prediction, and load imbalance.
- We introduce a method for assigning edge weights such that the cost of cutting a control-flow edge models the data dependence and thread misprediction overhead cycles incurred by placing a thread boundary on the edge.
- We present an abstract-execution-based scheme for comparing execution times of candidate threads.
- We have implemented the algorithm as part of a fully-automated, profile-based compilation process that measures 17 C and Fortran SPEC CPU2000 programs. Our method achieves an (geometric) average speedup of 74% for floating-point programs and 23% for integer programs, improving on the 52% and 13% achieved by the approach in [31].

In Section 2 we explain the execution model of a speculative CMP, followed by a discussion of our algorithm in Section 3 and results in Section 4. Additional related work is discussed in Section 5.

2. SPECULATIVE CMP EXECUTION MODEL

We introduce an execution model for speculative CMPs in terms of the execution overheads that are affected by the compiler's choice of thread boundaries. Our model is generally applicable to the architectures mentioned in Section 1. The primary difference among the architectures lies in the cache protocol they use for managing speculative storage and detecting misspeculation. Different cache protocols impact performance, but do not change the compiler's view of the execution model [3, 8, 9, 11, 28]. Thread-level speculation also has appeared in virtual machines [14]. The problem of partitioning a program into speculative threads arises with all of these architectures.

Thread Execution. A thread dispatcher (in hardware) fetches threads from the sequential instruction stream and dispatches them to processors. 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 interthread data dependence was violated. The CMP detects both types of violations and reacts by rolling back and restarting threads as necessary [7, 9]. The oldest thread in execution (w.r.t. sequential 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 [16].

Data Dependence Rollback 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. The reader and all younger threads are rolled back as in Figure 1. The entire 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. Furthermore, register dependences are specified by the compiler, allowing the hardware to communicate register values from one thread to another as appropriate [2]. CMP architectures also have evolved to learn and synchronize dynamically any frequently-encountered memory dependences that impede parallel execution [19].

Control Dependence Rollback Overhead. Control dependence rollbacks are caused by thread misprediction. A control dependence violation is detected when an older thread completes and its actual successor differs from the predicted successor. The overhead is the run time of all rolled-back, younger threads as in Figure 1.

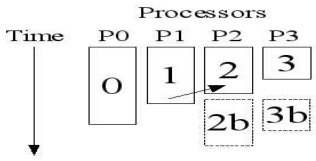

Figure 1: ***Rollback due to a data or control dependence violation:*** **Thread numbers indicate sequential order. Thread 1 detects a violation in thread 2. Thread 2 and the younger thread 3 are rolled back, followed by the dispatch of new threads on P2 and P3. Threads 2b and 3b may or may not be the 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 the processors in a cyclic order and a processor does not receive a new thread until it has committed its current thread. Because threads commit in program order, later threads have to wait for previous 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. Maintaining a cyclic dispatch order allows the sequence of threads to be determined easily for rollback operations. Although this order simplifies the architectural design, 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. Such hardware, however, may impact the access speed of the memory hierarchy. We do not assume such hardware in this paper.

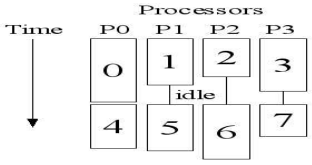

Figure 2: ***Load imbalance:*** **Strict cyclic dispatch order inherent in the architecture leads to load imbalance.**

General 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 will reduce the significance of this overhead; however, much larger threads (i.e., thousands of cycles) may overflow the speculative storage because they will include more writes to memory. An overflow completely stalls speculative execution, until the nonspeculative thread completes and allows the next thread to become nonspeculative, freeing speculative storage. Storage overflow occurs only in very large threads that access many distinct memory locations, and was not an issue in our benchmarks. Techniques exist to reduce the amount of speculative storage required by a program [8, 16].

3. COMPILER ALGORITHM FOR THREAD DECOMPOSITION

3.1 Compiler's Definition of a Thread

The compiler creates threads by designating control-flow graph (CFG) edges as *thread boundaries*. A thread begins at the first basic block of a procedure or after any thread boundary. The set of basic blocks in a thread is defined as the set of blocks reachable from its starting block without crossing a thread boundary. More formally, for a CFG $G = \{V, E\}$, where vertices represent basic blocks, the compiler determines boundary edges $E_b \subseteq E$ such that:

- $v \in V$ begins a thread iff $\exists e \in E_b$ s.t. $e = (u \in V, v)$ or v is the initial block of a procedure.
- $w \in V$ is part of the thread beginning at v iff $v \rightsquigarrow w$ in the graph $\{V, E_{\overline{b}}\}$, where $E_{\overline{b}} = E - E_b$.

A CFG with some cut edges is shown in Figure 3. Conveniently, each thread can be uniquely identified by its first basic block. A basic block is part of one or more threads and the compiler back end replicates blocks as necessary to package code into threads. Function calls are special: they can either be included or excluded from the calling thread. If *included*, all thread boundaries within the callee are ignored at run time; the hardware creates the illusion that the compiler had chosen $E_b = \emptyset$ for the CFG of the callee. Thus, inclusion prevents thread boundaries within function

calls from interrupting the calling thread, allowing the calling thread to contain more basic blocks at run time. If *excluded*, thread dispatch will continue with the threads in the callee. Figure 4 demonstrates inclusion versus exclusion and the effect on the number of run-time threads. This mechanism is our only means of making context-sensitive decisions for function calls, since thread boundaries within a function body are the same no matter the location from which it was called. The decision to include or exclude a call is made statically at each call site on the CFG. Hence, the entire function can execute as part of the calling thread, or execute as a fixed set of threads. Thread boundaries within a loop apply to all iterations of a loop.

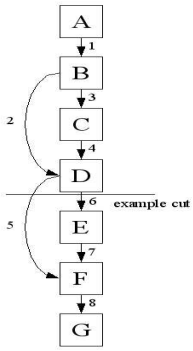

Figure 3: ***Relationship between basic blocks, threads, and edges:*** $E_b = \{e_5, e_6\}$ **while** $E_{\overline{b}} = \{e_1, e_2, e_3, e_4, e_7, e_8\}$**. Basic blocks A, E, and F begin new threads. Thread** T_A **contains blocks A, B, C, and D;** T_E **contains blocks E, F, and G; and** T_F **contains blocks F and G.**

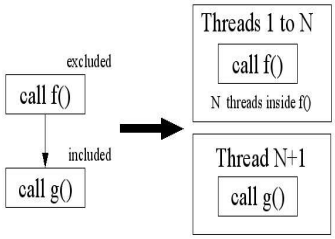

Figure 4: ***Function call exclusion versus inclusion:*** `f()` **is excluded, so the threads it contains are executed;** `g()` **is included, so the call and everything below it on the call graph executes as a single thread.**

3.2 Approach

3.2.1 Goal

The goal of our algorithm is to maximize the potential parallelism exposed to the underlying speculative CMP architecture. The architecture exploits the parallelism when it is actually present and ensures correctness when it is not. The compiler needs to attend carefully to each of the overheads introduced in Section 2 while designating CFG edges as thread boundaries. Edge weights are related to thread size and must change after each partitioning step. In our compiler implementation, thread boundaries are chosen after all other optimizations have been applied.

3.2.2 Min-Cut Decomposition

We apply a sequence of *balanced min-cuts*, where cut edges represent thread boundaries, to split a sequential program into threads. We use the standard min-cut [5, 6] for creating a cut, and an edge's weight represents the estimated cycles lost to misspeculation (both data and control flow) if the edge were cut. Thus the *cut-metric* is misspeculation penalties, and min-cut minimizes the penalties. By itself, however, the minimum cut does not ensure good parallelism; certainly in the extreme case, not cutting anything has zero penalty, but there would be no parallelism. To improve parallelism using balancing, we modify the algorithm originally proposed in [35]. In [35], balanced min-cut was applied to circuit placement and routing. Each balanced min-cut works by performing an initial min-cut and then reducing a second metric, the *balancing-metric*, which in [35] is the difference of the sum of vertex sizes of the two partitions. The cost of the cut will increase or remain the same while the balancing-metric is being reduced, because the initial min-cut is minimal. We make two important changes to this algorithm to use it for thread decomposition:

1. Our balancing-metric has two components: The first is the *estimated overall run time and cycles lost to load imbalance* assuming no misspeculation penalties. The second is the cut-metric that represents misspeculation penalties. The balancing-metric is the sum of the two components. Thus, our balancing-metric represents the overall run time spent by the threads for execution, load imbalance and misspeculation (i.e. performance).
2. Because our balancing-metric includes the cut-metric, our algorithm terminates when it reaches a minimal overall run time. By contrast, the balancing-metric does not include the cut-metric in [35]. That approach will continue considering more-balanced cuts even when the misspeculation penalty has increased to the point that it negates the performance gain from parallelism.

By using overall run time as the balancing-metric, our algorithm gives equal consideration to all the overheads at every balanced min-cut step, allowing one overhead to be traded off for another. By beginning with a sequential program and decomposing, our algorithm tends to select threads that are, on average, larger than the threads selected by [31], which builds threads bottom-up from basic blocks.

3.2.3 Differences from the Heuristic Approach

There are several differences between our approach and the approach in [31]. In [31], the overheads of prediction and data dependence are considered separately; the prediction heuristic overrides the dependence heuristic; and there is no consideration of load imbalance. Figures 5 and 6 show some problems that arise when the overheads are considered separately and are not given equal importance.

3.3 The Algorithm

3.3.1 Input

Our algorithm's input is an annotated CFG, where each vertex represents a basic block of the program. The annotations include static information about instructions per block. They also include dynamic information about branch frequencies, average cycles per function call, and dependences.

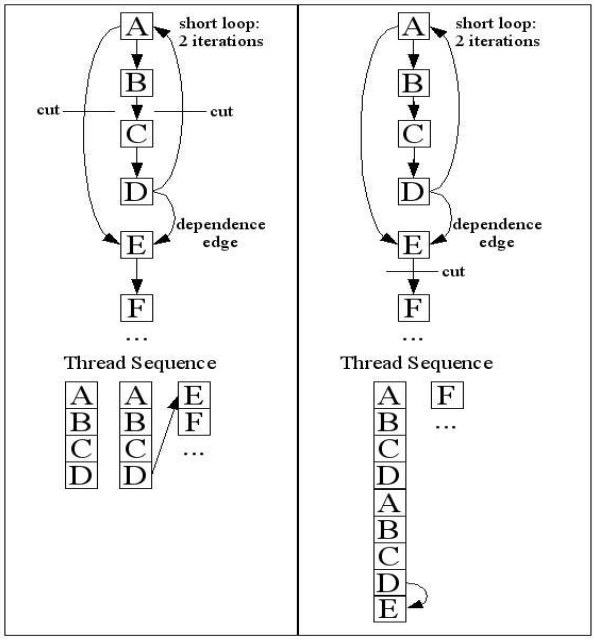

Figure 5: *Some differences between [31] (left) and our algorithm (right):* **On the left, the dependence heuristic suggests grouping basic blocks D and E into the same thread; however, the prediction heuristic overrides that decision assuming loops are highly predictable. Because the loop iterates only twice, the cost of cutting the dependence edge outweighs the gain from parallelizing the loop. On the right, the loop and block E are kept within a single thread, avoiding inter-thread dependence.**

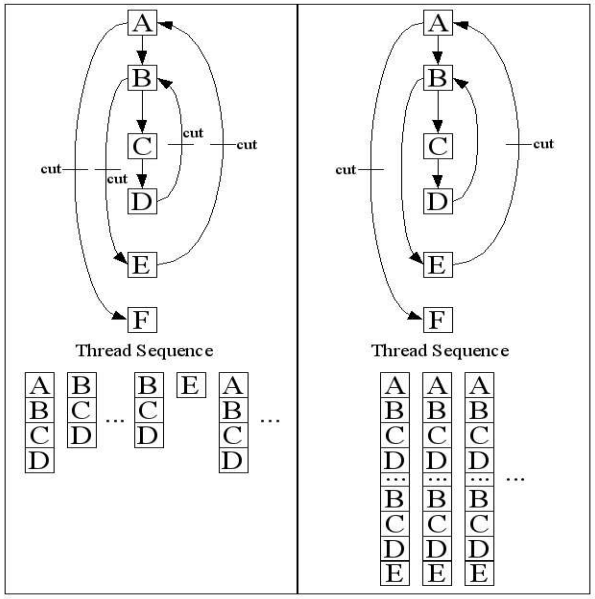

Figure 6: *Some differences between [31] (left) and our algorithm (right):* **On the left, the prediction heuristic causes threads to begin at all loop boundaries, even when there is coarser parallelism. These boundaries cause load imbalance, which will negate the speedup from parallelizing the inner loop unless the inner loop is large. On the right, inner loop iterations become part of the outer-loop thread, avoiding load imbalance.**

We do not claim that our profiling methods, described in Section 3.6, are perfect. We show that our algorithm improves performance using reasonable profiling information, and would expect equivalent or better performance with more accurate profiling.

3.3.2 Overall Scheme

A high-level view of the algorithm is shown in Figure 7 and psuedocode for the algorithm is shown in Figure 8. The following subsections will cover subroutines of the algorithm in detail. We begin with the entire program as a single thread. Each step seeks to decompose a procedure's CFG such that performance will improve. Initially, each thread includes (as in Section 3.1) its called subroutines. The initial part of the algorithm prepares the graph and the main part iteratively (re)computes the weights representing overheads and applies balanced min-cuts. As mentioned above, the first min-cut step minimizes the dependence and prediction penalties, while the balancing extension to min-cut reduces the overall run time. The $\mathcal{M}$ function represents the performance metric, where $\mathcal{M}(G) = \mathcal{R}(G) + \sum_{E_b} \mathcal{W}(e_i)$. $\mathcal{R}$ is the estimated run time including load imbalance, but excluding other penalties. $\mathcal{W}$ is the weight of an edge representing dependence and prediction penalties, $\mathcal{W}(e_i) = \mathcal{D}(e_i) + \mathcal{P}(e_i)$.

A number of balancing steps follow the initial cut. The algorithm tries moving each vertex bordering the cut to the other side, one at a time, and keeps the version with the best performance metric. After each vertex is moved, a new min-cut is performed. This new min-cut is necessary because the vertex move may cause the cut to cross an expensive edge. The authors of [35] show that these additional min-cuts do not increase the asymptotic complexity of the algorithm, provided that the number of vertex moves is bounded by some constant. They also show how to prevent min-cut from simply finding the same cut again and ignoring the move: the vertices on one side of the cut are temporarily collapsed into a *supernode*. Figure 9 shows the collapsing idea from [35]. Our balancing steps repeat while performance improves. Finally, our algorithm compares the best version to the performance of not cutting at all. If there is no improvement, decomposition of this thread stops. Otherwise, the best cut is applied and the newly-created threads are considered for decomposition in the same manner.

A key feature of our algorithm is that it is able to handle loops and straight-line code in the same way. Loops may become part of a single thread, each iteration may become a thread, or an iteration may get split up. The algorithm sees large weights across back edges of loops with cross-iteration dependences, biasing min-cut toward more independent loops with smaller weights. For nested loops that have identical edge weights, min-cut first encounters the outermost back edge as part of its normal operation. Therefore, coarser parallelism is tried before finer parallelism. Making both outer and inner loops into threads generally is avoided for perfectly nested loops because that causes load imbalance, which negatively impacts the performance metric. Together, these properties cause trade-offs that are more appropriate for handling loops than employed by [31], as shown in Figure 6.

Because our approach is a heuristic for an NP-complete problem, our algorithm may converge on a local optimum. Although using simulated annealing or genetic algorithms may overcome this limitation, such options would increase compilation time substantially. Instead, we use a simple perturbation in the final step, *edge perturbation*, to help free the algorithm from local optima.

We stipulate that every edge in the graph should be considered for cutting at least once. Each edge that was not examined throughout the sequence of balanced min-cuts is now examined exactly once to determine if cutting it improves performance. These edges are examined in reverse order, from the end of the procedure's CFG up to the entry point, such that any unexamined back edges of outer loops are encountered before inner loops. We choose to cut an edge if run time decreases. We consider each edge in isolation, so this perturbation step is linear in edges and does not increase our algorithmic complexity.

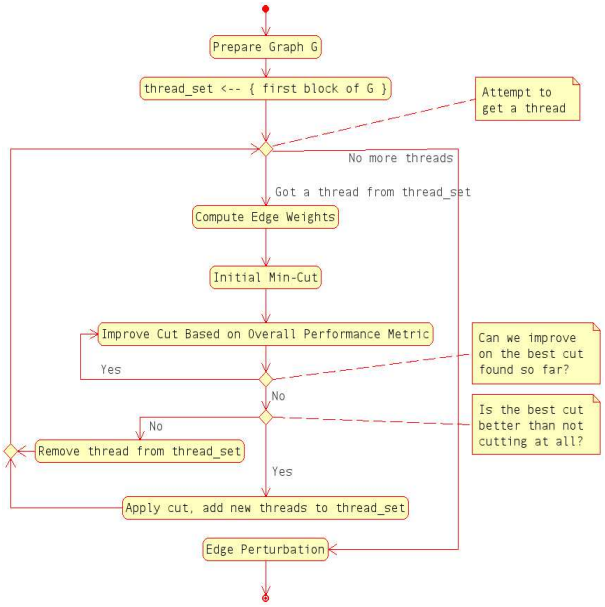

Figure 7: ***Our approach:*** **The edge-weight assignment and the performance metric calculation make use of profile information. A different metric could be substituted to accommodate different architectures or improved models.**

3.3.3 *PrepareGraph*

Our algorithm relies on the standard max-flow/min-cut algorithm [5, 6] to partition a set of vertices into subsets, such that the sum of the weights of the edges joining the subsets is minimal. Min-cut is not appropriate for a CFG containing loops however, because we often want to cut only the back edge of a loop without cutting its body. Min-cut is forced to cut at least two edges to break the cycle [34]. Therefore we use a technique called *vertex splitting*, shown in Figure 10, that replaces one vertex with two, and divides the edges of the original vertex between them. The transformation converts the back edge to a forward edge, while preserving the number of edges and all information necessary to use min-cut for penalty minimization. Only for

```
∀ procedure P ∈ Program
  G ← PrepareGraph(P)
  thread_set ← { GetEntryBlock(G) }

  while (thread_set ≠ ∅)
    /* Determine penalties for current threads in G */
    ComputeWeights(G)

    T ← GetFirstItem(thread_set)
    G_T ← {V_T, E_T}

    /* Initial bipartition of V_T */
    cut ← MinCut(G_T)

    /* Seek improvement */
    do
      initial_cut ← cut

      ∀u ∈ V_T s.t. (u, v) ∈ E_T or (v, u) ∈ E_T,
      and u and v are in different partitions
        G'_T ← temporarily collapse u with
          all vertices in v's partition
        temp_cut ← MinCut(G'_T)
        if M(G with temp_cut) < M(G with cut)
          cut ← temp_cut
    while cut ≠ initial_cut

    /* Improvement? */
    if M(G with cut) < M(G)
      G ← G with cut
      thread_set ← thread_set ⋃ new_threads
    else
      thread_set ← thread_set - T

  Edge Perturbation: Check all unexamined edges and
    cut if M(G with edge cut) < M(G)
```

Figure 8: ***Thread decomposition algorithm:*** **The goal is to find the set of boundary edges E_b with the best performance metric $\mathcal{M}(G) = \mathcal{R}(G) + \sum_{E_b} \mathcal{W}(e_i)$.**

purposes of estimating execution time is it necessary to remember that a loop was there, as illustrated by the dashed edge.

3.3.4 *ComputeWeights*

The dependence and prediction penalties are based on the surrounding thread sizes, so the edge weights must be recomputed as decomposition progresses to reflect new knowledge about the current set of threads. Each edge is assigned a weight that models the number of cycles that may be lost if a new thread were to begin at the sink of the edge. Weights are based on thread sizes, which change, and worst-case scenarios for wasted cycles. All edges have an initial weight to which dependence and prediction penalties are added. The initial weight is five cycles, a typical thread dispatch overhead for speculative CMPs. The dependence and prediction penalties are described in Sections 3.3.7 and 3.3.8, respectively. Thread sizes are computed as follows.

3.3.5 $\mathcal{S}$: Thread Size

$\mathcal{S}(T)$ is the size of thread T in terms of number of cycles. T may contain several possible control paths. Hence, the dynamic size of the thread depends on the actual path taken. Using profiling information, we compute an expected value for the dynamic size of T using equation (1). We represent the probability that path k is taken by p_k and the

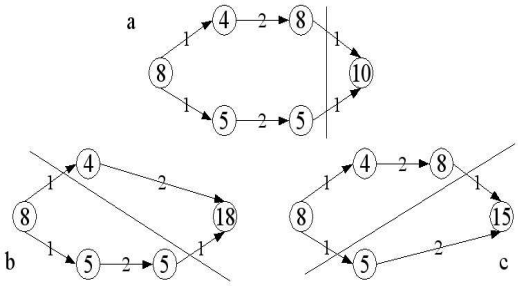

Figure 9: ***Balanced Min-Cut from [35]:*** **① Beginning with graph a, a min-cut is computed, resulting in a 30 to 10 split. ② The vertices that border the cut, of size 8 and 5, alternately are collapsed along with all vertices on the other side of the cut, leading to graphs b and c, respectively. ③: Min-cuts are recomputed. At this point the algorithm would choose c over b because it is more balanced, and then repeat Step ②. Applications of this algorithm use a balancing threshold and stop when the cut is balanced sufficiently. Note that the cut cost will remain the same or increase as balancing improves.**

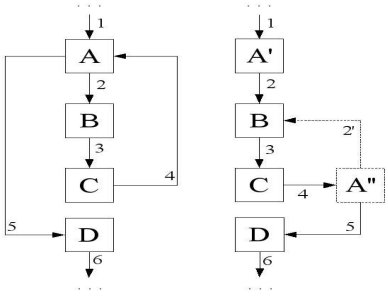

Figure 10: ***Vertex splitting:*** **Vertex A is temporarily split into vertices A' and A" while preserving edges 1-6. The dashed edge does not exist for penalty minimization, but is used for estimating execution time.**

size of path k as s_k. The probability of a path is the product of the branch frequencies along the path. The size of the path is computed as the sum of static instructions for basic blocks and dynamic cycles for function calls included along the path, similar to estimating procedure execution time in [24]. For loops, the total run time is the loop body run time multiplied by the average number of iterations.

Thread size can be computed efficiently by a depth-first traversal of the CFG, provided that loops are detected and handled properly. The expected thread size is the size of the initial block bb_0 plus the size of its target paths. Equation (1) gives the computation for any number of targets. For each recursive step, the target sizes are weighted by the probability of a particular branch being taken instead of the product of all branch frequencies along the path. That product will emerge naturally from the recursion. When a loop's back edge is encountered, the size of the subpath that was taken through the loop is multiplied by the average number of iterations minus one because the loop body has already been seen once (not shown in equation). Figure 11 shows the size calculation for a simple thread.

$$\mathcal{S}(T) = \sum_{k=1}^{paths} p_k s_k = bb_0 + \sum_{m=1}^{targets} p_m \mathcal{S}(m) \qquad (1)$$

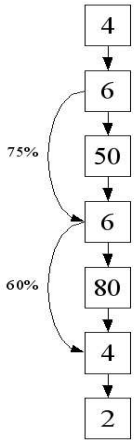

Figure 11: ***Expected value of the size of a thread:*** **There are four paths through the thread with sizes of 22, 72, 102, and 152. Their probabilities, respectively, are 0.45, 0.15, 0.30, and 0.10. The direct calculation gives $22*0.45+72*0.15+102*0.30+152*0.10 = 66.5$. The indirect, but more efficient, recursive method calculates the same result as $4+6+0.75t_1+0.25(50+t_1)$, where $t_1 = 6+0.6t_2+0.4(80+t_2)$, and $t_2 = 4+2$. The values t_1 and t_2 are computed once and reused.**

3.3.6 $\mathcal{R}$ and $\mathcal{L}$: Run Time and Load Imbalance

The expected run time $\mathcal{R}(G)$ of a graph G represents the first component of the balancing-metric, used in the min-cut balancing step. $\mathcal{R}(G)$ and the load imbalance $\mathcal{L}(G)$ are calculated by determining the run time and load imbalance of each possible sequence of threads and performing a weighted sum based on the probability of each sequence, as in (2), where p_k is the probability of each sequence s_k of threads, $r(s_k)$ is the run time of each sequence, and $l(s_k)$ is the load imbalance of each sequence. Enumerating all sequences would require exponential time. We approximate by considering up to n most likely sequences, where n is several hundred. The *abstract execution* algorithm in Figure 12 is used to find $r(s_k)$ and $l(s_k)$ under the assumption that no rollbacks occur. Since run time factors in load imbalance, $\mathcal{L}(G)$ is not explicitly added to $\mathcal{M}(G)$, but we show how it could be computed here. Potential rollbacks are not considered because they are modeled by the edge weights. A subsequence of threads that constitute a path through a loop has its expected run time multiplied by the average number of loop iterations and divided equally across the processors. This computation is more efficient than executing all threads from all iterations, especially in the case of nested loops, and produces nearly the same effect. Replacing the subsequence with these *num_procs* threads allows outer loops to be handled the same way. Time spent in an excluded function call (as per Section 3.1) is approximated by the performance metric that was computed for that function (computed on demand, thus enforcing a decomposition order on procedures), or simply the profiled execution time for recursive calls. The excluded call is expanded to *num_procs* threads, each the size of its performance metric, to repre-

```
prev = 0; imbalance = 0;

for (i = 0; i < num_procs; ++i)
   t[i] = 0;

for (i = 0; i < num_threads; ++i)
   if (loop detected)
      replace loop subsequence with num_procs threads
      each of size (subsequence_size *
      average_iterations / num_procs) and modify
      num_threads appropriately

   if (excluded call detected)
      replace call with num_procs threads
      each of size M(call) and modify
      num_threads appropriately

   p = i mod num_procs;
   if (t[p] < prev)
      imbalance += prev - t[p];
      t[p] = prev;
   else
      prev = t[p];

   t[p] += S(T_i);

r(s_k) = max(t[0], ..., t[num_procs-1]);
l(s_k) = imbalance;
```

Figure 12: ***Code for abstract execution of a thread sequence:*** **The if-else statement enforces the dispatch condition.**

sent processors being occupied by threads within the call. This approximation allows us to avoid executing the entire program interprocedurally.

$$\mathcal{R}(G) = \sum_{k=1}^{paths} p_k r(s_k),\ \mathcal{L}(G) = \sum_{k=1}^{paths} p_k l(s_k) \qquad (2)$$

3.3.7 $\mathcal{D}$: Dependence Penalty

$\mathcal{D}_i$, the dependence penalty due to cutting edge e_i, is estimated from dependence profile information. If a thread boundary must be placed across a dependence, the probability of it causing a rollback becomes greater the nearer it is placed to the sink.[1] Cutting just prior to the sink of a lexically forward dependence places the sink early in a thread and does not allow sufficient time for the source to execute. Similarly, cutting just after the sink of a lexically backward dependence places the source late in a thread, which has the same effect. It is not sufficient to begin a thread at a block that is not a sink, as suggested by [1], because the following block could be the sink of many dependences with sources late in the previous thread. It is also possible for innocuous interthread dependences to exist if the source will always execute before the sink. Therefore, our model for dependence penalties considers interthread dependence *distances*, instead of simply the number of interthread dependences.

In our model, data-dependence edges do not exist as separate edges in the CFG. We assign the data-dependence penalty to the edges along the appropriate paths of the CFG. An edge along a path from a source A to a sink B has its weight increased by the number of cycles that the sink would need to be delayed in order to satisfy the dependence if that edge were made a thread boundary, as in Figure 13. The delay is due to the synchronization mechanism [19] mentioned in Section 2, and is useful for modeling the "severity" of the dependence. If the quantity $X - Y$ is negative, then no penalty is applied. The penalty value is a worst-case estimate, as the actual delay depends on the thread start times, which are not known.

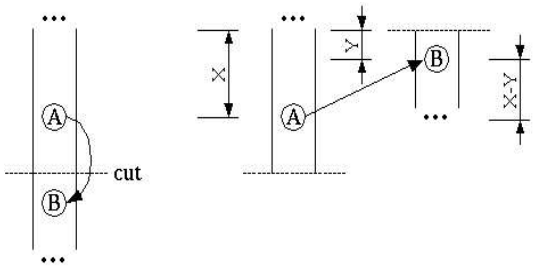

Figure 13: ***Effect of cutting a dependence edge from A to B:*** **B needs delayed by $X - Y$ to satisfy the dependence. Lexical backward dependences are handled similarly.**

3.3.8 $\mathcal{P}$: Prediction Penalties

$\mathcal{P}_i$, the thread prediction penalty due to cutting edge e_i, is estimated from branch frequencies and thread sizes. The hardware must predict the successor of each thread. If the prediction is incorrect, then the chosen successor will eventually be rolled back and cycles wasted. When thread boundaries are placed in regions of unpredictable control-flow, it is more difficult for the hardware to correctly predict the next thread. We use the simple approximation that a completely biased branch (0% or 100% taken) is perfectly predictable, while a completely unbiased branch (50% taken) is totally unpredictable. In between, we assume that predictability is linear. Equation (3) allows us to compute an expected value for the number of cycles lost. This value is added to the weight of each edge of the branch. There are better ways of modelling the hardware thread predictor; although we look at each branch in isolation, we could consider control-flow dependence information and use conditional path probabilities. We leave such extensions as future work.

$$\mathcal{P} = thread\_size * (1 - 2 * |0.5 - bfreq|) \qquad (3)$$

3.4 Algorithm Efficiency

Balanced min-cut is bounded by the cost of a single balancing step [35], which requires $O(VE^2)$ for the flow computation, $\Theta(E(V+E))$ to set prediction penalties, $\Theta(D(V+E))$ to set dependence penalties where D is the number of dependences between basic blocks, and $O(nT)$ for abstract execution up to a fixed number n of most likely paths (several hundred). Repeating balanced min-cut until there are T threads leads to a $O(VE^2T + D(V+E)T + nT^2)$ upper bound for decomposing a procedure. Perturbation at the end requires an additional $O(nET)$. In practice, $T \ll V$ and $V \approx 0.75E$, so the algorithm has a complexity of $O(V^3)$ assuming $D = O(V^2)$.

3.5 Compiler Implementation

Our compiler is based on the GNU C compiler and has been extended several times to add and improve support for

[1] In this section, sink refers to a dependence sink, and not the sink of a flow graph.

speculative CMPs. The f2c program allows the compiler to handle Fortran 77. Compilation involves two passes. The first pass performs common optimizations and unrolls loops with small bodies. The output of the first pass is a CFG for each source file. Our algorithm reads this graph, annotates it with dynamic profiling data, inserts thread boundaries as described in Section 3.1, and outputs new CFGs that reflect its decisions. GCC does not normally keep the CFG of the entire program in memory at once, but we save the CFG for each source file so that we have that information. The second pass of the compiler reads the new CFGs and generates Multiscalar code. Threads are placed in the binary according to the thread boundaries specified on the CFG. The binary is executed on a simulator to determine performance and generate the statistics discussed in Section 4. The binary could be executed on a real CMP, but currently there are no commercial speculative CMPs.

Two issues arise with interprocedural operations. One caveat is that `libc` calls must be included as in Section 3.1, because the library code does not contain thread headers. However, these calls are short and should be included for performance reasons anyway ([31] has the same restriction). The other problem is that calls through pointers represent an unknown, variable number of cycles because a single call site may invoke numerous different functions with different behaviors. The decision to include or exclude applies identically to all of these invocations. We force exclusion for two reasons. (i) Excluding these calls generally gave better performance than including them. (ii) We observed that programs typically call their own functions through pointers, instead of `libc` functions. While calls through pointers are not found in Fortran 77 programs, they are present in some C benchmarks (notably gap, mesa, and ammp) and would be more common in C++ programs. There is no ideal static solution to handling these calls, and we do not assume the hardware supports conditional run-time inclusion. We note that [31] is far more restrictive and excludes *all* non-library calls.

3.6 Obtaining Profile Information

We use profile information for dependencies and branch probabilities. We developed a source-code instrumentation tool to facilitate the run-time detection of data dependences. The tool uses a method similar to [22], which gathers dependences and dependence distances. The distances are important for handling loops. For example, a dependence crossing more iterations than there are processors is implicitly enforced by the execution model. Anti and output dependences are irrelevant, as explained in Section 2. The instrumentation prevents register allocation so it sees all dependences. Source code instrumentation also provides branch frequencies, however the profile may not cover all paths taken in the final run. For branches lacking this information, we use an estimate that 60% of forward branches are taken and 85% of backward branches are taken [12]. As is typical for profile-guided optimizations, the input data for the profile run differs from the input data used to evaluate our techniques [33]. For all profile runs, we use the `train` data set, and for all final runs we used the `ref` data set.

4. PERFORMANCE ON SPEC CPU2000

We evaluated the performance of our algorithm using the Multiscalar simulator, configured according to Table 1. Currently there are no commercial speculative CMPs; however, Multiscalar shares many properties with proposed implementations, and the memory system parameters we use are similar to an IBM Power4 [13]. We execute beyond each program's startup code with a functional simulation, before performing a detailed timing simulation for at least 500 million instructions. Rather than using a fixed number of instructions for detailed simulation, we use the same start and end points in the source code for all versions of a program. This methodology is important, because the instruction count of the different versions may vary due to the number and location of thread boundaries. To verify each benchmark passes the validation test, we complete the runs using the functional simulation. A typical simulation runs for one to three days.

SPEC CPU2000 contains 19 benchmarks written in C or Fortran 77, of which we use 17. Table 2 shows our benchmarks. We could not compile two C benchmarks, gcc and crafty, because our compiler is based on an old version of gcc. The rest are Fortran 90 codes that f2c cannot handle or contain C++.

Table 1: 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
Memory Latency	120 cycles

4.1 Performance and Overhead Analysis

Table 2 shows the baseline instructions-per-cycle (IPC) of each benchmark run as a single-thread on one processor, the speedup on four processors over the baseline using the techniques in [31], and our speedup over the baseline. Each processor is dual-issue out-of-order. Single-thread IPC is typically in the 0.50 to 1.50 range, though mcf is 0.23 and ammp is 0.12 due to poor cache behavior. In FP2000, applu, mgrid, and swim achieve a speedup above 2.0 using [31], while the rest of the benchmarks remain below a speedup of 1.40. Of particular interest are the INT2000 benchmarks, of which most remain below a speedup of 1.30. Integer (i.e., non-numerical) programs are known to be more difficult for compilers to parallelize than scientific programs, due to the lack of large, regular loops. Our results show similar trends, but note that a speculative CMP is able to extract some parallelism from these programs. The last two columns in the table show that in general our approach creates longer threads compared to [31], confirming that longer threads achieve more parallelism and better performance.

Table 2: Baseline vs Improved Performance

SPEC FP 2000	Single Thread IPC	[31]'s Speedup	Our Speedup	[31]'s Insns/ Thread	Our Insns/ Thread
ammp	0.12	1.02	1.04	14.2	15.5
applu	1.00	2.14	2.20	36.7	46.2
art	0.48	1.93	2.14	15.7	19.3
equake	1.23	1.07	1.15	15.5	19.6
mesa	1.56	1.13	1.34	33.2	49.0
mgrid	1.11	2.13	2.15	113.9	115.4
sixtrack	1.17	1.15	1.23	44.3	52.9
swim	0.58	2.57	2.93	101.9	112.8
wupwise	1.22	1.36	2.49	26.6	1374.7
g. mean		1.52	1.74		
SPEC INT 2000	Single Thread IPC	[31]'s Speedup	Our Speedup	[31]'s Insns/ Thread	Our Insns/ Thread
bzip2	1.37	1.08	1.12	12.7	16.6
gap	1.32	1.18	1.27	25.5	30.1
gzip	1.33	1.26	1.23	19.4	43.4
mcf	0.23	0.87	1.26	7.4	13.0
parser	1.14	0.98	1.02	10.3	11.3
twolf	1.05	1.09	1.12	12.4	15.2
vortex	1.46	1.33	1.45	15.7	24.9
vpr	1.30	1.36	1.47	26.0	32.2
g. mean		1.13	1.23		

Figure 14 shows the improvement gained by our decomposition method over [31]. To isolate the effect of the changing the edge weights as described in this paper, we show our method using changing weights (black bars) and using fixed weights (white bars). We first discuss the black bars. Mesa and wupwise have the best improvement. The INT2000 codes that show the most improvement are gap, mcf, vortex, and vpr. The average gain over [31] is 14.3% for floating-point programs and 9.0% for integer programs. A more detailed analysis showed that most of the improvement in FP2000 was from our algorithm using the dependence profile to identify potentially parallel outer loops. While our approach gives equal consideration to all the overheads, [31] gives precedence to the thread prediction heuristic which creates thread boundaries at *all* loops even when creating threads only from an outer loop would perform better. For wupwise, a significant outer loop was parallel, which lead to a large improvement. It is far more difficult to develop source-code-level explanations for improvement in the INT2000 programs, due to more complex control-flow. Nevertheless, we observe a trend that thread dispatch overhead and load imbalance usually decrease, while dependence and prediction overhead usually remain about the same. We infer there was potential for larger, more-balanced threads, but that dependence and prediction problems are still difficult to avoid at a coarser level. We verified our algorithm performs as well as manual thread selection for some parts of the smaller benchmarks.

Figure 14 also shows the effects of fixing weights at the beginning, as would be done by related scheduling algorithms. This variant of the algorithm assumes an average thread size and does not update the weights as decomposition proceeds. We note that the scheduling algorithms were not designed for our problem and are shown here merely to isolate the effect of our strategy of changing the edge weights. Our real point of comparison is [31] which directly targets our problem. Although it might be possible to find useful fixed weights for some programs through much trial and error, we believe these results show that recomputing weights is of paramount importance.

Figures 15 and 16 show the relative importance of the different overheads for each benchmark, the left bar (labeled as "a" below the X-axis) using [31] and the right bar (labeled as "b" below the X-axis) using our algorithm. The 100% mark represents the total amount of overhead for [31] for each benchmark. Comparing [31] to our approach, our overall overhead decreases even though some individual overhead increases. This behavior follows from the fact that by giving equal consideration to all the overheads, our algorithm trades-off one overhead for another to reduce the overall overhead. For instance, misprediction overhead for applu and mesa in Figure 15 and for bzip2 and gzip in Figure 16 increases though their overall overhead decreases. Similarly load imbalance overhead for swim in Figure 15 and for parser and vortex in Figure 16 increases. We make a few other observations. The poor cache behavior of ammp shows up as memory latency and dependence overhead. Load imbalance decreases in applu, equake, mesa, mgrid, and especially wupwise. Dependence is the overhead that typically is reduced for FP2000 benchmarks. Load imbalance and dispatch overhead typically is reduced for INT2000 benchmarks.

Overheads not directly related to speculation include memory latency and function-unit stalls. The overheads appear because both approaches are designed to target speculation overheads, and leave these problems for other optimizations, such as high-level source transformations or instruction scheduling within threads. Although memory latency is not modeled in our algorithm, our decomposition changes memory latency as a side effect due to changes in access locality. Two accesses to the same address separated into different threads, and hence different caches with [31], may be combined into one larger thread, and hence the same cache, with our algorithm. There is no support for forcing particular threads onto certain processors. Our thread sizes are generally larger, as can be seen in Table 2, and larger threads combine more nearby basic blocks which exhibit high data locality.

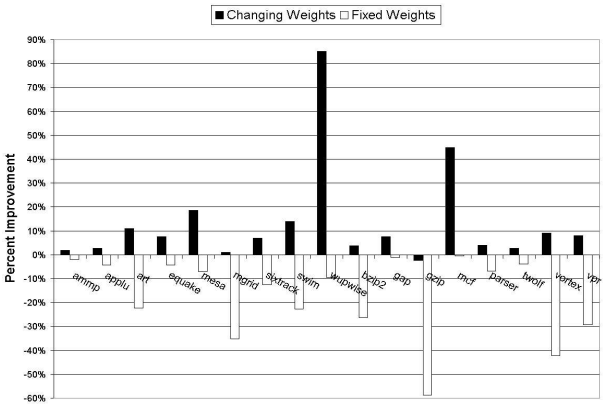

Figure 14: SPEC CPU2000 Improvement of our min-cut approach with changing weights over [31]. Assuming fixed weights, as in related scheduling algorithms, leads to dismal performance.

4.2 Compilation Time

We record in Table 3 the time required to compile each benchmark on a 450 MHz Sparc using min-cut decomposi-

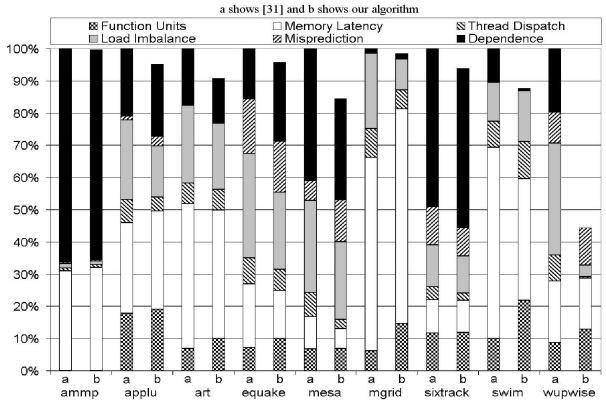

Figure 15: SPEC FP2000 Relative importance of reduced overheads compared to 100% of [31]'s overhead.

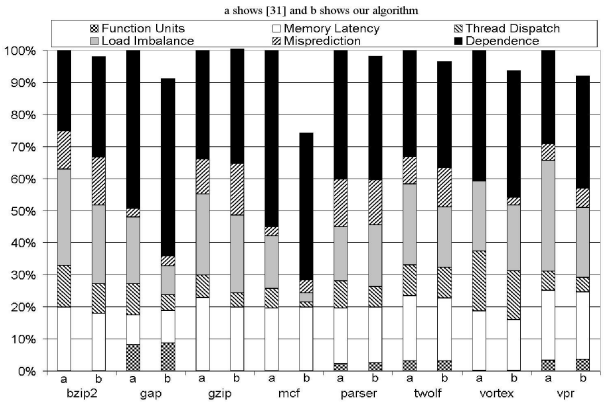

Figure 16: SPEC INT2000 Relative importance of reduced overheads compared to 100% of [31]'s overhead.

tion (MC) versus using the heuristics in [31]. Eight of the benchmarks finished in less than one minute, while 3-10 minutes was common for the rest. Sixtrack contains three extremely large procedures (`daten`, `maincr`, and `umlau6`) that each have over 2000 basic blocks and require a long time for either method. Due to the nature of our algorithm, a large program with typically small functions will decompose faster than a smaller program with several very large functions.

5. RELATED WORK

We have already discussed the relation of our work to scheduling. Our work differs from another recent compiler framework for speculative execution [1] in that we do not handle loops separately, address all overheads in an integrated fashion, and evaluate larger applications using more realistic hardware assumptions. Other compilers for CMPs have focused exclusively on creating threads from loop iterations or function calls, while performing much of the work manually [27]. Some compilers [29] are responsible for adding extra code to support speculation and synchronization, depending on the level of architectural support, whereas we are concerned only with thread boundary placement. Source-code transformations designed specifically to improve speculation, such as those applied manually in [20, 23], can be applied before performing thread selection. We assume that all transformations have been applied prior to decomposition, although we did not modify any of the source code for this paper. If source code is unavailable, it is still possible to make an existing binary run on a speculative CMP by annotating the binary with thread boundaries [17]. The partitioning can be done entirely in hardware, but suffers from a lack of compile-time information and increased run-time overhead [4]. Some compilers [15] consider both implicit and explicit (i.e., with speculation disabled) parallelism, if the target architecture supports both options [21].

Table 3: Mean and Maximum Vertices and Edges and Elapsed Time for Heuristics versus Min-Cut

FP2000	Lines	V_{avg}	E_{avg}	V_{max}	E_{max}	[31](s)	MC(s)
ammp	13483	35	48	492	649	144	284.3
applu	4975	62	82	169	232	186	232.1
art	1270	32	44	137	200	10	9.3
equake	1513	27	35	220	295	22	24.0
mesa	58724	23	33	786	1166	394	511.2
mgrid	1270	30	38	72	99	11	2.6
sixtrack	89918	112	146	2996	4018	3542	4983.1
swim	907	30	38	73	88	9	1.2
wupwise	3353	36	48	300	443	33	16.9
INT2000	Lines	V_{avg}	E_{avg}	V_{max}	E_{max}	[31](s)	MC(s)
bzip2	4649	24	33	328	465	24	43.4
gap	71363	43	62	1015	1554	412	607.2
gzip	8616	28	38	181	254	40	45.6
mcf	2412	27	39	83	128	12	13.1
parser	11391	22	30	343	426	61	203.8
twolf	20459	61	87	451	710	283	377.6
vortex	67213	25	35	614	916	307	491.5
vpr	17729	24	32	337	431	76	214.2

6. CONCLUSIONS

The issue of program decomposition for speculative execution arises in all speculative CMPs. We have presented an algorithm for decomposing programs into threads to expose parallelism to a speculative CMP, while considering multiple performance overheads related to data dependence, load imbalance, and thread prediction. The key challenge is that the overheads depend on the thread size, and change as decomposition progresses. Our algorithm uses a sequence of balanced min-cuts, which gives equal consideration to all the overheads, and adjusts the edge weights after every cut. We have compared our scheme with the previous heuristic method developed for the Multiscalar architecture. While the previous method uses each heuristic to target an individual overhead in isolation and gives precedence to the thread-prediction heuristic, our algorithm provides a more integrated solution by simultaneously considering all the overheads. Compared to other work, we have an effective, automated decomposition method for programs instead of only loops. Our results show an average speedup of 48% across SPEC CPU2000, whereas the multiple-heuristic approach yields an average of 32%.

7. REFERENCES

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

[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. 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.

[4] 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.

[5] J. Edmonds and R. M. Karp. Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. *Journal of the ACM*, 19:248–264, 1972.

[6] L. R. Ford Jr. and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.

[7] 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.

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

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

[10] L. Hammond, B. A. Nayfeh, and K. Olukotun. A Single-Chip Multiprocessor. *IEEE Computer*, 30(9):79–85, 1997.

[11] 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.

[12] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach, 3rd Edition*. Morgan Kaufmann, New York, 2002.

[13] IBM. IBM e-Server Power4 System Microarchitecture. Technical report, October 2001.

[14] I. H. Kazi and D. J. Lilja. JavaSpMT: A Speculative Thread Pipelining Parallelization Model for Java Programs. In *Proceedings of the 14th International Parallel and Distributed Processing Symposium*, pages 559–564, May 2000.

[15] S. W. Kim and R. Eigenmann. The Structure of a Compiler for Explicit and Implicit Parallelism. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing (LCPC)*, August 2001.

[16] S. W. Kim et al. Reference Idempotency Analysis: A Framework for Optimizing Speculative Execution. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, volume 36, pages 2–11, 2001.

[17] V. Krishnan and J. Torrellas. Hardware and Software Support for Speculative Execution of Sequential Binaries on a Chip-Multiprocessor. In *Proceedings of the International Conference on Supercomputing*, pages 85–92, July 1998.

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

[19] 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.

[20] 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.

[21] C. L. Ooi et al. Multiplex: Unifying Conventional and Speculative Thread-Level Parallelism on a Chip Multiprocessor. In *Proceedings of the International Conference on Supercomputing*, June 2001.

[22] P. M. Petersen and D. A. Padua. Static and Dynamic Evaluation of Data Dependence Analysis. In *Proceedings of the International Conference on Supercomputing*, pages 107–116, July 1993.

[23] M. K. Prabhu and K. Olukotun. Using Thread-Level Speculation to Simplify Manual Parallelization. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, June 2003.

[24] V. Sarkar. Determining Average Program Execution Times and their Variance. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 298–312, 1989.

[25] 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.

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

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

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

[29] J. G. Steffan and T. C. Mowry. The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization. In *Proceedings of the 4th IEEE Symposium on High-Performance Computer Architecture*, pages 2–13, February 1998.

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

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

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

[33] D. W. Wall. Predicting Program Behavior Using Real or Estimated Profiles. In *Proceedings of the Conference on Programming Language Design and Implementation*, volume 26, pages 59–70, June 1991.

[34] D. B. West. *Introduction to Graph Theory - Second Edition*. Prentice-Hall, 2001.

[35] H. H. Yang and D. F. Wong. Efficient Network Flow Based Min-Cut Balanced Partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(12), December 1996.

[36] 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, 1998.