Experiences with Helper Threads for Speculative Parallelization in Software

Marcelo Cintra¹ and Diego R. Llanos²

¹ School of Informatics, University of Edinburgh, Edinburgh, UK
mc@inf.ed.ac.uk
² Departamento de Informática, Universidad de Valladolid, Valladolid, Spain
diego@infor.uva.es

Abstract. Software schemes to speculative parallelization require no changes to the hardware of existing shared-memory systems, but can suffer from significant overheads. This paper evaluates the use of helper threads to assist the execution of the compute threads by performing some of the most expensive speculative operations on their behalf. Performance gains can be achieved if alleviating such overheads compensates for the loss of processing resources available for the main computation. Moving speculative operations from the main threads to helper threads, however, is not trivial and introduces several new race conditions, which are addressed in this paper. Experimental results so far show that for an SMP system helper threads fail to materialize any significant performance gains, but that for a system with multithreading a more promising outcome can be expected.

1 INTRODUCTION

Current parallelizing compilers still fail to parallelize codes when data dependence information is incomplete. In these cases, run-time parallelization in software has been explored with the speculative parallelization technique [3, 7, 8, 12, 13]. With this technique the code is speculatively executed in parallel while the reference stream is monitored for data dependence violations. If a dependence violation is found, the system reverts the state back to some safe condition and threads are re-executed. While various degrees of hardware support for speculative parallelization have been proposed, these are costly and require modifications to the processors and caches. In this paper, we focus on software-only implementations of speculative parallelization. In this case, the user application itself is augmented with code to perform all the speculative operations, which introduces significant overheads.

The contribution of this paper is to evaluate the use of helper threads to assist the execution of the compute threads by performing some of the most expensive speculative operations on their behalf. The key insight is that for many applications allocating processing resources to alleviate the speculative operations

* This work was supported in part by the European Commission under grant RII3-CT-2003-506079.
overheads may more than compensate for the loss of processing resources available for the main computation. Moving speculative operations from the compute threads to helper threads, however, is not trivial and introduces several new race conditions that must be addressed. In this paper we describe how to extend an existing software speculative parallelization scheme [3] and address the arising issues and races. The issues that we address and the ideas behind our solutions are general and applicable to other software speculative parallelization schemes.

Experimental results so far show that for an SMP system with 4 or 8 processors the performance degradation of sacrificing a processor for a helper thread cannot be overcome, and with 16 and 24 processors the reduction of overheads through the helper thread mechanism is minimal and leads to no significant performance gains. However, initial experiments with a multithreaded system with 4 processors show that helper threads have a better potential to lead to performance benefits.

The rest of this paper is organized as follows: Section 2 briefly describes speculative parallelization and the software scheme that we build on; Section 3 presents our proposed helper thread scheme and discusses in detail the implementation issues; Sections 4 and 5 describe our evaluation methodology and experimental results; Section 6 discusses related work; and Section 7 concludes the paper.

2 SPECULATIVE PARALLELIZATION

Under speculative parallelization (also called thread-level speculation or speculative multithreading) threads are extracted from sequential code and run in parallel, hoping not to violate any sequential semantics. The control flow of the sequential code imposes a total order on the threads. At any time during execution, the earliest thread in program order is non-speculative while the others are speculative. The terms predecessor and successor are used to relate threads in this total order. Stores from speculative threads generate unsafe versions of variables, while loads from speculative threads return potentially incorrect versions.

As execution proceeds, the system tracks memory references to identify any cross-thread data dependence violation. If a violation is detected, the offending thread must be squashed, along with its successors. When a thread is squashed, all the data that it speculatively modified must be purged from the memory hierarchy and the thread then restarts execution from its beginning. During re-execution the thread can be then provided with the updated value.

When the execution of a non-speculative thread completes it commits and the versions it generated can be moved to safe storage. At this point its immediate successor acquires non-speculative status and is allowed to commit. If a speculative thread completes it must wait for all predecessors to commit before it can commit. In some cases, however, the processor associated with this speculative thread is allowed to start execution of a new speculative thread while the commit is pending. After committing, the processor is always free to start executing a new speculative thread.
2.1 A Software-only Speculative Parallelization Scheme

In this section we briefly describe the design of the baseline scheme for software speculative parallelization that we build on. The reader is referred to [3, 5] for more details.

Speculative Access and Window Structures To manage speculative threads, the scheme implements in software a sliding window mechanism that allows for larger number of uncommitted threads than the number of processors in the system. The sliding window mechanism is implemented as an array of characters of length $W$ containing a status descriptor for each uncommitted (active) thread slot, which says whether there is no thread, a running thread, a squashed thread, or a finished but uncommitted thread associated with the slot. Additionally, two integers mark the boundaries of the window at any time, pointing to the non-speculative and the most-speculative threads.

To support both fast commits and fast checks for data dependence violations upon memory accesses, the scheme uses a set of three speculative access structures. The first structure, called $AM$ or Access Matrix, is an $M \times W$ array of characters, where $W$ is the window size and $M$ is the size of the user data structure. Each element in this speculative access structure encodes the following four states: not accessed data ($\text{NotAcc}$), exposed loaded data ($\text{ExpLd}$), modified data ($\text{Mod}$), and exposed loaded and modified data ($\text{ExpLdMod}$). The second speculative access structure, called $IM$ or Indirection Matrix, is an $M \times W$ array of integers where the first elements in each column point to elements of $AM$ in states other than $\text{NotAcc}$ and the last valid element is identified by a tail pointer. The scheme also uses a third access structure, called $G\text{ExpLd}$ or Global Exposed Load. It is a single array of $M$ logical values that indicates whether at least some thread, since the start of the speculative execution, has performed an exposed load to this particular datum.

Speculative Memory Operations The scheme supports forwarding of predecessor versions, so upon a speculative load the $AM$ data structure is searched backward (i.e., toward the non-speculative thread) for entries in states $\text{Mod}$ or $\text{ExpLdMod}$. Search for data dependence violations are performed on every speculative store, and require scanning the $AM$ data structure forward for entries in states $\text{ExpLd}$ or $\text{ExpLdMod}$, or until an entry in state $\text{Mod}$ if found. Note that squashes can only be triggered by stores. In addition to speculative loads and stores the scheme also implements a speculative reduction operation.

Commits The code executed at the completion of a thread is divided in two main sections: the commit proper and the assignment of a new thread. Only the non-speculative thread performs commits, and it is responsible for committing itself and all successor threads that have already finished. Committing involves scanning the $IM$ and $AM$ data structures of the threads being committed for entries in states $\text{Mod}$ or $\text{ExpLdMod}$ and then writing the corresponding values in
the version copies back to the original user data structure. When the window is full, processors spin-wait until a thread slot is freed.

**Squashes** The squash operation in the scheme simply involves setting the window state of the violating thread and all its successors to squashed and moving the most-speculative pointer backward. This puts those threads back in the pool of threads to be executed and, later, when the squashed threads finish they will be re-executed. The costly aspect of the squash operation is that it has to be performed within a critical section.

### 3 MOVING SPECULATIVE OPERATIONS TO HELPER THREADS

The first step in designing a speculative parallelization engine based on helper threads is to identify operations that are likely to be profitably moved to helper threads. Good candidate operations must satisfy two properties: be time-consuming and not lie in the critical path of the speculative execution. The critical path of the speculative execution consists of the actual computation in the non-speculative thread and, to a lesser extent, in the speculative threads. Referring to the operations discussed in Section 2 we identify the following candidate operations: (1) squash, (2) search for violations in speculative store, and (3) search for predecessor versions in speculative load. The completion of these operations is not required to allow progress of the actual computations of the threads. We note that these operations are required by any speculative parallelization scheme, are expensive in all, and are not simply artifacts of our particular implementation. We discuss the implementation of helper threads for each of these operations in turn in Sections 3.2 to 3.4.

In this paper we do not consider dynamically spawning helper threads on demand, but, instead, have a fixed number of helper threads throughout execution, which run on dedicated processors. In our implementation, each helper thread services a fixed subset of the compute threads and each compute thread can have only one outstanding request, blocking when they attempt to place a second request.

#### 3.1 New Data Structures

We implement the interface between compute and helper threads with a simple new data structure to enqueue requests. This data structure is composed of the following fields for each window slot:

- **Window ID:** contains either the id of the thread that requested the operation or the id of the first thread to be squashed.

- **Speculative data index:** contains the index of the speculative data structure that is involved in this particular operation.

- **Position of non-speculative pointer:** contains the position of the non-speculative pointer (required to guarantee correctness of forwarding, as explained in Section 3.4).
Access flag: contains a boolean used to synchronize the helper thread and compute thread. Each compute thread is given a single access flag.

Decoupled operation: contains the opcode of the operation requested.

Helper threads cyclicly check the access flags of all compute threads assigned to them until they find one that is set. Compute threads, on the other hand, instead of performing the speculative operations themselves now spin-wait on their respective access flag and enqueue the request once the flag is unset. Care must be taken to guarantee that the parameters passed through the data structure are globally visible before setting the flag.

3.2 Squash Operation

The squash operation can take much useful time from compute threads as it requires access to the critical section, but moving this operation to helper threads would only benefit applications that suffer frequent data dependence violations. Because this operation requires access to the critical section there is no benefit from having multiple helper threads. An optimization that is possible with squashes is to merge multiple squash requests since a squash of thread $i$ subsumes a squash of $i + j$. This can be done by letting helper threads determine the first (least speculative) thread that must be squashed by checking all the pending squashes, and not only the one they were asked to perform.

Moving the squash operation to a helper thread introduces new race conditions between a squash request and a commit. Consider the following sequence of events: speculative thread $i$ ($spec_i$) places a squash request starting at $spec_j$; before the helper thread has a chance to process this squash all speculative threads between the non-speculative thread ($nonspec$) and $spec_j$, inclusive, complete execution; then $nonspec$ completes execution, assumes that all threads up to $spec_j$ have safely completed, and (incorrectly) commits their versions to the reference storage. This race did not exist in the original protocol because $spec_i$ would have to process the squash itself before completing execution.

Our solution to this race condition is to introduce two requirements. First, enqueued squash requests must be made visible to all threads before the requesting thread is allowed to proceed with computation. Second, before a thread is allowed to commit it must check that there are no pending squashes. This test must be performed after the non-speculative thread enters the critical section to commit and, to avoid deadlocks, the non-speculative thread must release the critical section if the test fails, so that the helper thread may process the pending squashes. Note that livelock is possible if the committing thread repeatedly enters the critical section, fails to commit, and releases the critical section, and must be avoided with some fairness mechanism. These two requirements, combined with the original protocol’s restriction that speculative threads must also get access to the critical section in order to complete, guarantee that the commit will never include versions of threads that may still be squashed.

One optimization that we employ is to relax the requirement that no squashes can be pending while a thread commits. We note that threads that have pending squash requests and that have not yet completed execution will not be committed
(since they will not be able to complete during another thread’s commit), and
neither will their squashed successors. Thus, we can still proceed with the commit
of threads as long as all threads that have completed and would be candidates
for being committed do not have any pending squash requests. Only if any such
thread has a pending squash request will the non-speculative thread have to exit
the critical section.

3.3 Search for Violations

Searching for data dependence violations can become very time consuming be-
cause it requires testing several elements of AM and some of these are likely to
be coherence cache misses, since they are set by other threads. Thus, the orig-
inal store operation, which is usually handled out of the critical path through
write buffers in the processor, can now incur several load misses, which is likely
to affect the critical path. The optimization with the GIExpLd data structure
helps when applications do not generate exposed loads, but even in this case the
access to GIExpLd is likely to incur an extra cache miss.

This operation does not require access to a critical section and, thus, multiple
helper threads can service such requests simultaneously. If the helper thread
detects a data dependence violation it simply enqueues a squash request to be
handled later. To avoid having helper threads enqueue requests themselves, we,
instead, reuse the current request from the compute thread changing it from a
check for violations to a squash request. Finally, moving the search for violations
to a helper thread leads to no new race conditions.

3.4 Search for Predecessor Versions

Searching for a predecessor version can become very time consuming because
it requires testing several elements of AM and some of these are likely to be
coherence cache misses, since they are set by other threads. Thus, the original
load operation, a single miss in the worst case, can now incur several load misses.

Different from the other speculative operations, moving the search for prede-
cessor versions from the speculative load operation to a helper thread requires
a major change in the protocol. This is because the original protocol supports
forwarding (Section 2), so that the result of the search is needed as the return
value of the speculative load. Thus, we must change the protocol to not support
forwarding, so that a speculative load will always return the current value stored
in the original user data structure. Then the search for a predecessor version be-
comes a check for a data dependence violation: if such an interveining version is
found a data dependence was violated and the thread must be squashed. This
change will mostly affect applications that have data dependences, in which case
some successful forwardings may turn into additional squashes.

This operation does not require access to a critical section and, thus, multiple
helper threads can service such requests simultaneously. If the helper thread
detects a data dependence violation it simply enqueues a squash request to be
handled later. To avoid having helper threads enqueue requests themselves, we,
instead, reuse the current request from the compute thread changing it from a check for predecessor version to a squash request.

Moving the search for predecessor versions to a helper thread introduces new race conditions between a search for predecessor version and a commit. Consider the following sequence of events: speculative thread $i$ ($spec_i$) performs a speculative store; $spec_j$, successor of $spec_i$, performs a speculative load, obtains the reference value, and places a check for a predecessor version request; before the helper thread has a chance to process this request all speculative threads between the non-speculative thread and $spec_i$, inclusive, complete execution and commit, writing the value produced by $spec_i$ back to the reference copy and advancing the non-speculative pointer to $i + 1$; then the helper thread processes the check for predecessor version on behalf of $spec_j$ and stops the search at the new value of the non-speculative pointer, thus, missing the version produced by $spec_i$ and (incorrectly) concluding that $spec_j$ has not caused a data dependence violation. Note that simply using the old value of the non-speculative pointer in the check is not enough since the speculative access information may have been reused and lost. This race did not exist in the original protocol because $spec_j$ would have to wait for the search for predecessor version to complete before using either such version or the reference copy.

Our solution to this race condition is to introduce three requirements. First, we make the committing thread partly responsible for detecting this race, and violation, when it writes back a predecessor version to the user data structure. This can be done by letting the committing thread check if there is a pending request for a check for predecessor version for every data that is written back. A pending check means that some thread has incorrectly used this reference value instead of the predecessor version. Second, the helper thread can also detect a possible race if it compares the value of the non-speculative pointer at the time of the speculative load against its current value. If the value has changed then an intervening commit has happened. This can be done by letting the speculative threads pass the current value of the non-speculative pointer to the helper thread when they request checks for predecessor versions. These two requirements are enough to guarantee that a violation is detected. However, we still have the problem that if a speculative thread finishes and is committed before all its pending checks for predecessor versions are performed then, even though the violation will be detected, it will be too late to perform the squash. Thus, a third requirement is that a thread is not allowed to complete until all its pending requests are handled by the helper threads; this is intuitive as, unlike the other operations, the outcome of this operation may affect the requesting thread itself.

4 EVALUATION METHODOLOGY
4.1 Applications

We use four applications: TREE from [1], WUPWISE from SPECfp2000 [15], MDG from the PERFECT Club suite [2], and 2D-HULL from [4]. These applications are representative of legacy as well as recent sequential scientific programs.
Table 1. Characteristics of the applications and input sizes used.

<table>
<thead>
<tr>
<th>Application (loop)</th>
<th>Input</th>
<th>Spec data size in KB</th>
<th>Iterations per invocation</th>
<th>% of violations</th>
</tr>
</thead>
<tbody>
<tr>
<td>TREE (accel_10)</td>
<td>Off-axis parab. collision, 4096 particles</td>
<td>&lt; 1</td>
<td>4096</td>
<td>0</td>
</tr>
<tr>
<td>WUPWISE (muldeo_200', muldoe_200')</td>
<td>reference input</td>
<td>12,000</td>
<td>8,000</td>
<td>0</td>
</tr>
<tr>
<td>MDG (interf_1000')</td>
<td>reference input</td>
<td>13</td>
<td>343</td>
<td>0</td>
</tr>
<tr>
<td>2D-HULL (main loop)</td>
<td>Square input set, 40M particles</td>
<td>15</td>
<td>39,999,997</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Disc input set, 40M particles</td>
<td>137</td>
<td>39,999,997</td>
<td>7</td>
</tr>
</tbody>
</table>

They spend a large fraction of their sequential execution time on loops that cannot be automatically parallelized by state-of-the-art compilers. The first three applications do not suffer from RAW data dependences at run time, while 2D-HULL does.

Table 4.1 shows, for each application, the loops that we attempt to parallelize speculatively, the different input sizes used, the size of the data accessed through speculative references, the average number of iterations executed per loop invocation, and the fraction of threads that suffer data dependence violations. In the case of WUPWISE, we obtain loops muldeo_200' and muldoe_200' by merging the three outer loops in loop nests muldeo_200 and muldoe_200, respectively.

The input sets used with TREE, WUPWISE, and MDG are the standard ones provided with the applications. For 2D-HULL we evaluate two different input sets that distribute the points inside a square or inside a disc. The shape of the distribution of input points affects the number of data dependences. For TREE, WUPWISE, and MDG we use a chunk size of two iterations and a window size of two [3]; for 2D-HULL we also use a window size of two, but we use a chunk size of 1024 and 5000 for the disc and square inputs, respectively.

4.2 Parallel Execution Environment

We have run the applications on two systems. The first is a 48-processor Sun Fire 15K symmetric multiprocessor (SMP) with 900MHz UltraSPARC-III processors and 48GBytes of shared main memory. The system runs SunOS5.8. The second is a 2-processor DELL PowerEdge 1750 SMP with 3.0GHz Xeon processors and 2GBytes of shared main memory. Each processor has two contexts with simultaneous multithreading (SMT). The system runs Linux 2.4. In both systems we used the native compilers with the highest optimization settings and we used OpenMP 2.0, because of its wide acceptance and portability, to parallelize the loops and to implement the memory fences required for correctness of the protocol. The applications had exclusive use of the processors during the entire execution and we use wall-clock time in our time measurements. For the
execution time breakdowns in the Fire we use the Forte performance collector tool, which introduced negligible execution overheads.

5 EXPERIMENTAL RESULTS

5.1 Baseline Speedups and Overheads

We start by discussing the speedups for the baseline system (no helper threads) and evaluating the main sources of overheads in the speculative parallelization code. The leftmost bar in each group in Figure 1 shows the execution time breakdown of the speculative sections only for the baseline scheme with $P$ compute threads ($PC$), with $P$ varying from 4 to 24 processors. The bars are normalized to the sequential execution time and are broken down into the following components: execution time of the original loop body plus OpenMP overhead ($Busy$); idle time due to load imbalance ($Spin$); overhead time spent on speculative loads and stores\(^1\) ($Memory$ ops.); overhead time of the commit operations and setting up of a new thread ($Commit$); and idle time waiting to enter the critical section to commit ($Contention$).

From the figure we observe that the baseline speculative parallelization scheme achieves only a fraction of the potential speedup, which is closely represented by the $Busy$ portion of the bars. It is clear that there is still a lot of room for improvement for the baseline scheme as the difference between the $Busy$ portions and the full bars range from 22% to 79% and is 43% on average. The main sources of overheads identified in Section 3, plus commit, are listed in Table 5.1.

From the table we observe that, collectively, the four sources of overheads highlighted amount to up to 37% of overhead, and 18% on average, which is a large fraction of the performance degradation of the baseline scheme. We also observe that no single source of overhead stands out as the main source for all applications: forwarding is by far the main source of overhead for 2D-HULL, commit is the main source of overhead for MDG, search for violations is the main source of overhead for TREE, and all three, forwarding, commit, and search for violations, are important sources of overhead for WUPWISE. Surprisingly, squash is a negligible source of overhead for 2D-HULL. This happens because in our implementation the squash operation is actually very efficient. For this reason and because of space restrictions we do not show results for helper thread scheme for the squash operation only. It suffices to say that, as expected, this scheme neither benefits 2D-HULL nor harms the applications without dependences.

Another important metric to understand the possible performance benefits or losses due to the use of helper threads is the performance difference between the baseline with $P$ processors and baseline with $P - H$ processors, where $P$ is the number of processors and $H$ is the number of helper threads being considered. We call this metric the *performance hurdle* that a helper thread scheme must overcome. Table 5.1 shows the performance hurdles with $H = 1, 2$. For $H = 1$ this

\(^1\) With helper threads this component also includes idle time waiting to enqueue a second request.
Fig. 1. Normalized execution time breakdowns. Results are shown for 4, 8, 16, and 24 processors. The numbers on top of the bars are the speedups relative to sequential execution.
<table>
<thead>
<tr>
<th>Application</th>
<th>Input</th>
<th>N procs.</th>
<th>Squash</th>
<th>Search Viol.</th>
<th>Forwarding</th>
<th>Commit</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>2D-HULL</td>
<td>Disc, 40M</td>
<td>4</td>
<td>0.0</td>
<td>0.0</td>
<td>10.0</td>
<td>1.3</td>
<td>11.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>0.1</td>
<td>0.0</td>
<td>18.0</td>
<td>1.1</td>
<td>19.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16</td>
<td>0.0</td>
<td>0.0</td>
<td>31.3</td>
<td>1.1</td>
<td>32.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>24</td>
<td>0.0</td>
<td>0.0</td>
<td>36.5</td>
<td>0.9</td>
<td>37.4</td>
</tr>
<tr>
<td></td>
<td>Square, 40M</td>
<td>4</td>
<td>0.0</td>
<td>0.0</td>
<td>14.0</td>
<td>0.4</td>
<td>14.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>0.0</td>
<td>0.0</td>
<td>14.8</td>
<td>0.1</td>
<td>14.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16</td>
<td>0.0</td>
<td>0.0</td>
<td>16.1</td>
<td>0.2</td>
<td>16.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>24</td>
<td>0.0</td>
<td>0.2</td>
<td>17.1</td>
<td>0.4</td>
<td>17.7</td>
</tr>
<tr>
<td>MDG</td>
<td>ref</td>
<td>4</td>
<td>0.0</td>
<td>1.2</td>
<td>0.0</td>
<td>7.2</td>
<td>14.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>0.0</td>
<td>1.3</td>
<td>0.0</td>
<td>6.2</td>
<td>7.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16</td>
<td>0.0</td>
<td>0.6</td>
<td>0.0</td>
<td>5.0</td>
<td>5.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>24</td>
<td>0.0</td>
<td>0.7</td>
<td>0.0</td>
<td>7.1</td>
<td>7.8</td>
</tr>
<tr>
<td>TREE</td>
<td>1024 part.</td>
<td>4</td>
<td>0.0</td>
<td>2.6</td>
<td>0.0</td>
<td>1.1</td>
<td>3.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>0.0</td>
<td>6.4</td>
<td>0.0</td>
<td>1.3</td>
<td>7.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16</td>
<td>0.0</td>
<td>10.2</td>
<td>0.0</td>
<td>1.1</td>
<td>11.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>24</td>
<td>0.0</td>
<td>13.3</td>
<td>0.0</td>
<td>0.7</td>
<td>14.0</td>
</tr>
<tr>
<td>WUPWISE</td>
<td>ref</td>
<td>4</td>
<td>0.0</td>
<td>4.0</td>
<td>7.5</td>
<td>4.9</td>
<td>16.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>0.0</td>
<td>5.5</td>
<td>13.2</td>
<td>3.9</td>
<td>22.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16</td>
<td>0.0</td>
<td>4.7</td>
<td>15.3</td>
<td>13.2</td>
<td>33.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>24</td>
<td>0.0</td>
<td>10.5</td>
<td>19.3</td>
<td>0.0</td>
<td>29.8</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td></td>
<td>0.0</td>
<td>2.2</td>
<td>13.3</td>
<td>1.9</td>
<td>18.3</td>
</tr>
</tbody>
</table>

Table 2. Speculative parallelization overheads.

metric can also be seen as the difference between the first and second leftmost bars in each group in Figure 1.

From the table we observe that the hurdles for both values of $H$ are very high for configurations with 4 processors. These hurdles are very similar to the speculation overheads, which means that helper threads are unlikely to benefit systems of this size. For larger configurations, on the other hand, the hurdles reduce significantly, even for $H = 2$.

5.2 Helper Thread for Search for Violations and Squash

The execution time breakdown of the helper thread scheme for the search for violations and squash operations combined is shown as the $(P-H)HC+HSV$ bar in Figure 1. From the figure we observe that for MDG the performance of this helper thread scheme is slightly better for 16 and 24 processors than that of the $(P-H)HC$ scheme and sometimes than that of the baseline. Since MDG has no data dependences and no exposed loads this performance improvement comes from time savings in the look-up to the GiExpLd structure.

For TREE and 2D-HULL the performance of this helper thread scheme is worse than that of the baseline. This performance degradation comes from no benefit of the helper operation and from the performance hurdles.
Table 3. Performance hurdles.

For WUPWISE the performance of this helper thread scheme is significantly worse than that of the baseline. From the overheads and performance hurdles one would not expect such behavior. The problem, however, is that the helper threads cannot cope with the requests generated by the compute threads. Each compute thread performs 1344 speculative stores and spends on average 37% of its time waiting to enqueue requests.

5.3 Helper Thread for Forwarding and Squash

The execution time breakdown of the helper thread scheme for the forwarding and squash operations combined is shown as the (P-H)C+HSF bar in Figure 1. From the figure we observe that for MDG the performance of this helper thread scheme is slightly better for 16 and 24 processors than that of the (P-H)C scheme and sometimes than that of the baseline. Since MDG has no data dependences this performance improvement comes from time savings in the search for predecessor versions, which always fails. For TREE the performance of this helper thread scheme is worse than that of the baseline. An important observation is that for both applications this helper thread scheme does not in practice generate many unnecessary squashes.

For 2D-HULL and WUPWISE the performance of this helper thread scheme is significantly worse than that of both the (P-H)C and the baseline. From the
overheads and performance hurdles these were perhaps the most promising combinations of applications and helper thread scheme. The problem, however, is that the helper threads cannot cope with the requests generated by the compute threads. Compute threads perform a large number of speculative loads in 2D-HULL and WUPWISE and spend on average 50% and 30% of their time waiting to enqueue requests, respectively. Also for these applications this helper thread scheme does generate many additional squashes, which is clear from the significant increase in the Busy portion. These extra squashes, in turn, increase the occupancy of the helper threads and require access to the critical section, which lead to further performance degradation.

5.4 Combined Speedups

The execution time breakdown of the helper thread scheme for all operations combined is shown as the (P-H)C+HSFV bar in Figure 1. From this figure we observe that the performance of the combined helper thread scheme is similar to the performance of the least performing helper thread scheme. This results in slight performance improvements for MDG with 24 processors, in slight performance degradation for TREE, and significant performance degradations for 2D-HULL and WUPWISE.

5.5 Simultaneous Multithreading

To assess the potential of the helper thread scheme on a system with simultaneous multithreading we compare the speedups of manually parallelized versions of TREE and MDG on both the Fire and the PowerEdge systems. Figure 2 shows the execution times and the hurdles. From the figure we observe that going from 3 to 4 processors gives much smaller performance improvements, and, thus, smaller hurdles, on the PowerEdge. This is as expected since this system has only 2 physical processors and the 4th thread is always contending for resources with the other three. At the time of this writing we have not yet been able to tune the speculative parallelization schemes for the PowerEdge/x86. However, this result indicates that with SMT systems the addition of a compute thread in some configurations leads to only marginal gains and, thus, helper threads may offer a better alternative.

6 RELATED WORK

Run-time speculative parallelization in software was introduced in the LRPD test [12], and several other schemes have been proposed since [7, 8, 13]. In [3, 5] we proposed and evaluated the baseline scheme used in this paper. In all these works the speculative operations are executed directly by the compute threads, and this work is the first to attempt to off-load some of these operations to dedicated helper threads. Several hardware approaches for speculative parallelization have been proposed (e.g., [9, 14, 16]). While these alleviate many of the overheads of speculative parallelization by moving some of the operations
to hardware, they require significant changes to the hardware structures, such as caches, protocol controllers, and even the processors. Finally, several works have investigated the use of helper threads to accelerate single-threaded execution. Many of these require extensive hardware support, but some are mostly software based (e.g., [6, 10, 11]). These works usually use helper threads to speculatively perform time-consuming critical-path tasks ahead of time in an attempt to accelerate the critical path. Our work applies helper threads in the context of speculative parallelization and uses them to perform off-critical-path tasks on behalf of the main compute threads.

7 CONCLUSIONS

In this paper, we proposed and evaluated the use of helper threads to assist the execution of speculatively parallelized applications. The key insights are that only a few protocol operations account for most of the overheads of speculative parallelization and that for large number of processors the performance improvement/degradation of having one more/less processor is very small (we called this performance difference the performance hurdle that a helper thread scheme must overcome). Moving speculative operations from the compute threads to helper threads, however, is not trivial and introduces several new race conditions that must be addressed. In this paper we described in detail how to extend a software speculative parallelization scheme with helper threads and addressed the arising issues and races. Experimental results show for an SMP system with 4 or 8 processors the performance degradation of sacrificing a processor for a helper thread cannot be overcome, and with 16 and 24 processors the reduction of overheads through the helper thread mechanism is minimal and leads to no significant performance gains. However, initial experiments with a multithreaded system with 4 processors show that helper threads have a better potential to lead to performance benefits in such systems.
References