Design and Implementation of A Compiler Framework for Helper Threading on Multi-Core Processors

Yonghong Song Spiros Kalogeropulos Partha Tirumalai
Scalable Systems Group
Sun Microsystems, Inc.
{yonghong.song, spiros.kalogeropulos, partha.tirumalai}@sun.com

Abstract

Helper threading is a technique that utilizes a second core or logical processor in a multi-threaded system to improve the performance of the main thread. A helper thread executes in parallel with the main thread that it attempts to accelerate. In this paper, the helper thread merely prefetches data into a shared cache and does not incur any other programmer visible effects. Helper thread prefetching has been proposed as a viable solution in various scenarios where it is difficult to prefetch efficiently within the main thread itself. This paper presents our helper threading experience on SUN's second dual-core SPARC microprocessor, the UltraSPARC IV+. The two cores on this processor share an on-chip L2 and an off-chip L3 cache. We present a compiler framework to automatically construct helper threads and evaluate our scheme on the UltraSPARC IV+ processor. Our preliminary results using helper threads on the SPEC CPU2000 suite show gains of up to 22% on programs that suffer substantial L2 cache misses while at the same time incurring negligible losses on programs that do not suffer L2 cache misses.

1. Introduction

With the widening gap between processor and memory speeds, prefetching has been increasingly important to improve application performance [16, 18, 25, 26]. Currently, prefetching is most effective for memory access streams where future memory addresses can be easily predicted using loop index values [7, 18, 19]. For such access streams, software prefetch instructions are inserted into the program to bring data into cache before the use. Such a prefetching scheme in which the prefetches are interleaved with the main computation is also called *interleaved* prefetching.

Although it is quite successful for many cases [7, 19], interleaved prefetching tends to be less effective for two kinds of codes. First, for codes with complex array subscripts, memory access strides are often loop variant, even predictable, at compile time. Prefetching in such codes tends to incur excessive overhead as significant computation is required to compute future addresses. The complexity and overhead increase if the subscript evaluation involves loads that themselves must be prefetched and made speculative. One such example is an indexed array access [7]. If the prefetched data is already in the cache, such large overheads can cause a significant slowdown. To avoid risking large penalties, modern production compilers often ignore such cases by default, or prefetch data speculatively, one or two cache lines ahead.

The second class of difficult codes involve pointer-chasing. In these codes, at least one memory access is needed to get the memory address in the next loop iteration. Interleaved prefetching is not able to handle such cases. Several techniques have been proposed to attack pointer-chasing. Luk and Mowry proposes several compiler algorithms for recursive data structures [14, 15]. Their approaches, however, do not solve prefetching for general pointer-chasing codes. The jump-pointer approach [21] requires whole program analysis which may not be possible at compile time. Cahoon and McKinley try to detect the regularity of the memory stream at compile time for Java applications [3]. Adl-Tabatabai *et al.* develop a runtime scheme in a JIT Java compiler, using hardware counters to identify delinquent loads and memory address profiles to identify regularity of memory access streams [1]. Such information is later used by the runtime system for prefetch instruction insertion. Wu tries to detect the regularity of the memory stream with value profiling [28]. The success of his algorithm depends on how *closely* the training and actual inputs match each other as well as on how many predictable memory streams exist in the program.

Chip multi-threading (CMT) architectures with shared caches present new opportunities for prefetching. With CMT, another core or logical processor may be used to prefetch data needed by the main thread. Helper thread-

Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)
1089-795X/05 $20.00 © 2005 IEEE
IEEE COMPUTER SOCIETY

ing is a technique which can perform such prefetching in software. A helper thread, which is created at runtime, executes in parallel with the main thread, and does not have any programmer visible side effects. In our context, the helper thread attempts to prefetch data accessed by the main thread into the shared cache. Since it does not do any computation or stores other than the computation necessary to form prefetchable addresses and maintain approximate (often exact) control flow, the helper thread will typically execute faster than the main thread and act as an effective prefetcher to the main thread.

Prefetching with helper threading naturally handles the cases where interleaved prefetching is ineffective. In codes involving complex array subscripts, prefetching overhead is *offloaded* to the helper thread. For pointer-chasing codes, helper threading tries to speculatively load or prefetch what could be actually cache missing. Helper threading, however, is not free. Launching the helper thread and synchronization between the main thread and the helper thread incur overhead. Such overhead must be minimized by the compiler as well as the runtime system.

In this paper, we make the following contributions:

- Detailed algorithms for helper threading region selection and code generation are presented. Our scheme is also implemented in a production compiler code base.
- We evaluate our scheme on real hardware, using an UltraSPARC(TM) IV+ processor based system and the SPEC CPU2000 benchmark suite [22]. The UltraSPARC IV+ processor has two on-chip cores and a shared on-chip L2 cache. We compare helper threading performance to the best up-to-date serial performance, and show that our helper threading improves performance by up to 22% for programs that suffer L2 cache misses while at the same time causing minimal degradation on most programs that don't suffer significant L2 cache misses.

The rest of paper is organized as follows. In Section 2, we describe the architecture of the UltraSPARC IV+ processor. In Section 3, we describe compiler support for helper threading. In Section 4, we discuss the runtime support needed for helper threading. We evaluate our implementation in Section 5, compare to previous work in Section 6, and draw a conclusion in Section 7.

2. Architecture Description

Our helper threading implementation targets the UltraSPARC IV+ microprocessor [6]. This chip has two 4-issue in-order superscalar cores each of which implements the functionality in a significantly enhanced UltraSPARC III design [11]. Each core has its own first level instruction and data caches, both 64KB. Each core also has its own instruction and data TLB's. The cores share an on-chip 2MB level 2 unified cache which has low latency and adequate bandwidth to support smooth dual core operation. Also shared is a large 32MB off-chip dirty victim level 3 cache. The level 2 and level 3 caches can be configured to be in split or shared mode. In split mode, each core can allocate only in half the cache. However, each core can read all of the cache. In shared mode, each core can allocate in all of the cache. Unless otherwise mentioned, the expermental data presented in this paper are all with the caches in shared mode. Figure 1 shows a simple block diagram of the UltraSPARC IV+ processor.

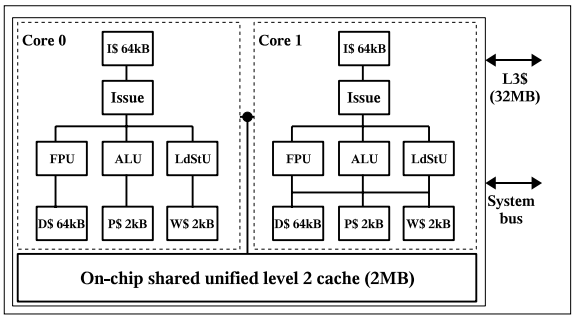

Figure 1. Block diagram of UltraSPARC IV+ processor.

The UltraSPARC IV+ processor implements the 64-bit SPARC V9 ISA [27] with extensions. With respect to our helper threading study, the implementation of the software prefetch extensions is important. Nine flavors of software prefetch are supported. These include the four flavors in SPARC V9: read once, read many, write once, and write many. These four variants can be either weak or strong. Weak prefetches are dropped if a TLB miss occurs during prefetch address translation. On the other hand, strong prefetches will generate a TLB trap and the prefetch will be processed (after the trap). An instruction prefetch is also provided for prefetching instructions. A control bit in the processor further controls the behavior of weak prefetches. When the 8-entry prefetch queue is full, either they can be dropped or they can stall the processor until a queue slot is available.

In our study, we allow the main thread to use all prefetch variants. Program analysis and compiler options determine the variants used for prefetchable accesses. Unless otherwise mentioned, the helper thread uses only strong prefetch variants. If prefetches were dropped on a TLB miss, the benefit of helper threading would be lost or vastly diminished. Our systems also had the prefetch control setting to disallow dropping of weak prefetches if the prefetch queue

1089-795X/05 $20.00 © 2005 IEEE
IEEE COMPUTER SOCIETY

```
/* The overall algorithm */
build loop tree hierarchy.
perform reuse and prefetch analysis.
/* Assuming root_loop is the root of the loop tree. */
call prefetching_using_helper_thread_driver (root_loop).
```

(a)

```
procedure prefetching_using_helper_thread_driver (Loop *loop)
   if (is_helper_thread_candidate (loop)
      and is_profitable_to_use_helper_thread (loop)) then
        helper_thread_code_gen (loop)
   else
      for (each immediate inner loop of loop, inner_loop) do
        prefetching_using_helper_thread_driver (inner_loop )
      end for
   end if
end procedure
```

(b)

Figure 2. The overall algorithm.

is full.

3. Compiler Support for Helper Threading

3.1. Overview

Our helper threading work focuses on loops. The compiler tries to analyze the program and identify the loop regions which are candidates for helper threading using the following criteria:

- The loop contains memory accesses which potentially could incur cache misses.
- The prefetches generated by the helper thread will trigger cache misses sufficiently early before the prefetched data are used by the main thread.
- It is profitable to use a helper thread to generate prefetches for the loop. The benefit from such prefetching outweighs the cost of using a helper thread.

Figure 2 shows the overall algorithm. In Figure 2(a), a loop hierarchy tree is first built for the whole program, followed by *reuse* analysis and prefetch candidate identification [7] to introduce only necessary prefetches and avoid issuing redundant ones. The function *prefetching_using_helper_thread_driver* is then called recursively to identify candidates and generate codes for helper threading. In Figure 2(b), if a loop in the loop hierarchy is identified as a helper threading candidate and it is profitable to use a helper thread, the loop will be transformed for helper threading purpose. Otherwise, its immediate inner loops will be examined further.

Due to the dynamic nature of operating system scheduling, the following two issues need to be addressed in code generation:

- Ensure the helper thread will do useful work.
- Avoid slowdown of the main thread.

The first issue is addressed by checking whether the main thread has completed the execution of the loop before the helper thread starts the execution of the corresponding loop, and by the helper thread inquiring periodically whether the main thread has completed the execution of the loop.

The second issue is addressed by avoiding synchronization with the helper thread at the end of the main thread for each particular helper threading loop, and by inserting prefetch instructions in the main thread as in the interleaved prefetching mode.

In the rest of this section, first, we present how loops are selected to be helper threading candidates. Then, we present our approach that determines if it is profitable to use a helper thread for a loop. Finally, the code generation scheme for helper threading is presented.

3.2. Selecting Candidate Loops

The benefits of using a helper thread for prefetching to speed up the main thread come from the following:

- The helper thread has potentially less computation to execute than the main thread. It could, therefore, execute certain loads earlier and bring their values to the shared L2 cache.
- Certain loads, if their loaded values are not used to compute a branch condition or an address used by another load/store, can be transformed into prefetches in the helper thread. Furthermore, stores can also be transformed into prefetches. These prefetches can bring data to the shared L2 cache, representing a potentially significant execution time saving for the main thread. The above load or store is called an *effective* prefetch candidate, if its address computation depends on at least another load in the same loop body or the load/store is identified as a prefetch candidate by using reuse analysis [7].

If the application is memory-bound, the first potential benefit will be less because the loads in both the main thread and the helper thread could be in the *critical* path for the application. Our scheme selects candidate loops mainly based on the second potential benefit. In the final code of the helper thread, all effective prefetch candidates will be replaced by strong prefetches to their corresponding addresses, in order to realize the potential benefit for the main thread.

Our compiler encodes alias information derived from pointer and array memory accesses in the data flow graph. The data flow generated by such alias information may be

IEEE COMPUTER SOCIETY

```
procedure is_helper_thread_candidate (Loop *loop)
  if (there exists any calls with side effects in the loop body)
  then
    return FALSE
  end if
  if (the loop is computation bound) then
    return FALSE
  end if
  if (there exists no effective prefetch candidate) then
    return FALSE
  end if
  for (all effective prefetch candidates and conditional branches)
  do
    if (if floating-point computation is required
        directly or indirectly) then
      return FALSE
    end if
  end for
  return TRUE
end procedure
```

Figure 3. The algorithm to select helper threading candidate loops.

```
procedure is_profitable_to_use_helper_thread (Loop *loop)
  p_overhead = startup_cost + parameter_passing_cost
  p_benefit = 0
  for (each effective prefetch candidate in the loop body) do
    p_benefit = p_benefit + num_of_accesses * L2_miss_penalty
      * predicted_L2_miss_rate
  end for
  if (both p_benefit and p_overhead are known at compile time) then
    if (p_benefit ≤ p_overhead) then
      return FALSE
    else
      return TRUE
    end if
  else
    /* two-version loops will be generated. */
    return TRUE
  end if
end procedure
```

Figure 4. The algorithm to determine the profitability of using a helper thread for candidate loops.

conservative and limit helper threading scope if it is required to maintain precise control flow and address computation in the helper thread. To overcome such limitation, the helper thread periodically checks whether the corresponding loop in the main thread is completed or not. This permits the compiler to ignore those conservative data flow edges and their *def-use* chains, when determining effective prefetch candidates and constructing final branch resolution codes. Although this could result in certain incorrect final prefetch addresses and incorrect control flow, such an omission enables more loops, especially more outer loops, as candidates in pointer-intensive programs. In particular, outer loop candidates tend to greatly increase the potential benefit for helper threading without increasing the cost much (see Section 3.3).

Figure 3 shows the algorithm to decide whether a loop is a helper threading candidate. Loops which contain function calls with side effects will not be considered as candidates. Furthermore, computation bound loops, which means that there is enough computation to hide memory latency, are not considered as candidates. Such an exclusion prevents cases with a heavy-weight main thread and a light-weight helper thread, where the helper thread may run too far ahead and overwrite useful data used in the main thread due to the limited size of the shared L2 cache. Finally, a candidate loop must have at least one effective prefetch candidate to ensure helper threading is potentially beneficial, and its effective prefetch candidates and conditionals do not contain floating-point computation to avoid potential exception.

3.3. Determining Profitability of Candidate Loops

The implementation of helper threading utilizes the existing automatic parallelization infrastructure which uses a fork-join model [23]. The parallelizable loop will be outlined and a runtime library is called to control dispatching the threads, synchronization, etc. Parallelization involves overhead in the runtime library and also parameter passing overhead due to outlining. The benefit of using a helper thread comes from the potential cache hit in the main thread for some memory accesses which could be cache misses in a single-threaded run. The compiler analyzes the potential benefit of using a helper thread versus parallelization overhead to decide the profitability of using a helper thread for a loop.

Figure 4 shows the algorithm to determine helper threading profitability for a candidate loop. The overhead of parallelization is computed as the runtime library cost, *startup_cost*, and the cost of passing various shared and first/last private variables [20], *parameter_passing_cost*. The *startup_cost* is a fixed empirical value and the *parameter_passing_cost* is the cost of passing the value for one variable, which is also a fixed empirical value, multiplied by the number of variables.

The computation of the helper threading benefit is focused on effective prefetch candidates. For each effective prefetch candidate, the potential saving, *p_benefit*, is computed as the total number of memory accesses in one invocation of this loop, *num_of_accesses*, multiplied by the L2 cache miss penalty, *L2_miss_penalty*, multiplied by the potential L2 cache miss rate for this memory access, *potential_L2_miss_rate*. The *L2_miss_penalty* is a fixed value given for a specific architecture. In the absence of cache profiling, our approach to determine the *potential_L2_miss_rate* value for an effective prefetch candidate is based on the complexity of its address computation and whether a prefetch is available in the main thread. The current values of *potential_L2_miss_rate* are determined experimentally for different address computation complexity levels.

1089-795X/05 $20.00 © 2005 IEEE

IEEE COMPUTER SOCIETY

The computation of the number of accesses for an effective prefetch candidate (*num_of_accesses*) depends on the availability of the profile feedback information. If the profile feedback information is available, the *num_of_accesses* is computed as the total number of memory accesses for an effective prefetch candidate divided by the times the *loop* is accessed, as the overhead is computed for each invocation (not each iteration) of the *loop* If the profile data shows that the *loop* is not accessed at all, the value for *num_of_accesses* is set to 0.

If the profile feedback information is not available, the value of *num_of_accesses* is computed based on the compile time information of loop trip counts and branch probability. If the actual trip count is not known at compile time, our approach is to examine whether the trip count can be computed symbolically through some loop invariants. Otherwise, a trip count of 25 will be assumed [10]. For conditional statements, equal probability for *if* taken/non-taken targets or all *case* targets of a *switch* statement is assumed. The total number of accesses, *num_of_accesses*, will be computed based on trip counts and assigned branch probability information.

The total benefit of using a helper thread for a loop, *p_benefit*, is the summation of the benefits of all effective prefetch candidates. if *p_benefit* is greater than *p_overhead* using compile time information, this loop will be a candidate for helper threading. Otherwise, if *p_benefit* is no greater than *p_overhead*, this loop will not be a candidate. Furthermore, if the compile time information produces inconclusive profitability result with symbolic trip count computation, a *two-versioned* loop with a runtime condition for profitability *p_benefit* $>$ *p_overhead* will be generated. At runtime, if the condition is true, the helper threading version will be executed. Otherwise, the original serial version will be executed.

3.4. Code Generation

Code generation for a candidate loop to use helper threading involves three phases. In the first phase, code like Figure 5(a) will be generated. The runtime library has been modified to guarantee that if the loop t is parallelized and two threads are available, the main thread will execute the branch if "$t == 0$" is true, and the helper thread will execute the other branch. The purpose is to minimize the overhead for the main thread to avoid the main thread slowdown. The helper thread may incur potential overhead to warm up its L1 cache and the TLB. The **else** branch loop in Figure 5(a) will be transformed to form a helper thread loop.

In the second phase, a proper helper thread loop will be generated through program slicing and variable renaming. The helper thread loop is a sliced original loop containing

```
for (t = 0; t <= 1; t = t + 1) {
 if (t == 0) {
  <The original loop>
 }
 else {
  <The original loop>
} }
```

(a)

```
for (t = 0; t <= 1; t = t + 1) {
 if (t == 0) {
  <The main thread loop>
 }
 else {
  <Scalar renaming assignments>
  <The helper thread loop with
   scalar renaming>
} }
```

(b)

```
is_mt_done = FALSE.
#pragma omp parallel for
for (t = 0; t <= 1; t = t + 1) {
 if (t == 0) {
  <The main thread loop>
  is_mt_done = TRUE.
 }
 else {
  if (is_mt_done == FALSE) {
   <Scalar renaming assignments>
   /* checking is_mt_done every certain
      number of loop iterations
      periodically. */
    <The helper thread loop with
     scalar renaming>
} } }
```

(c)

Figure 5. Transforming the original loop to a DOALL loop.

only the original control flow and necessary statements to compute conditionals and the effective prefetch candidate addresses. All effective prefetch candidates are replaced by *strong* prefetches to their corresponding addresses. In the helper thread, all loads will become *non-faulting* loads to avoid exceptions, and all stores will be either removed or turned to strong prefetches.

All *upward-exposed or downward-exposed* assigned variables in the helper thread loop will be renamed and copy statements of original variables to their corresponding temporary variables are placed right before the helper thread loop. In our scheme, all scalar variables are *scoped* as *private* variables including first private, or both first and last private (see Section 3.5), so that these temporary variables will get correct values at runtime. Figure 5(b) shows the code after program slicing and variable renaming.

In practice, it is possible that the helper thread could run behind the main thread. If this happens, the helper thread should finish early to avoid doing useless work. In the last phase, the following code is inserted to ensure that the helper thread is terminated when it runs behind the main thread.

- Code to indicate that the main thread loop has completed execution immediately after the main thread loop.
- Code to check whether the main thread loop has completed execution before executing the helper thread loop.
- Code to check whether the main thread has completed execution every certain number of loop iterations in the helper thread loop and all its inner loops. This can be done by adding checking at every loop back edge,

IEEE COMPUTER SOCIETY

```
procedure helper_thread_code_gen (Loop *loop)
  /* step 1: generated unsliced helper thread loop */
  make a copy of the original loop and generate code
    like Figure 5(a).

  /* step 2: program slicing and variable renaming for
     the helper thread loop. */
  for (each effective prefetch candidate in
     the helper thread loop) do
    mark all statements for its address computation, directly
      or indirectly, as undeletable.
    turn this load or store to a strong prefetch, and mark it
      as undeletable.
  end for
  for (every branch in the loop body) do
    mark all statements for branch condition computation,
      directly or indirectly, as undeletable.
    mark this branch as undeletable.
  end for
  delete all the unmarked deletable statements.
  for (every upward-exposed or downward-exposed variable v
    in at least one assignment in the helper thread loop) do
    create a temporary variable tv and an assignment tv = v
      right before the helper thread loop.
    rename all appearances of v with tv in the
      helper thread loop body.
  end for
  /* The code like Figure 5(b) is generated. */

  /* step 3: insert checking code to prevent the helper thread
     from running behind the main thread. */
  add an assignment right after the main thread loop to
    indicate it has completed execution.
  add a check whether the main thread loop has completed execution
    or not before executing the helper thread loop.
  add code at every back edge of the helper thread loop and
    its inner loops to check whether the main thread has
    completed execution or not.
  make the loop t in Figure 5(c) as DOALL loop and perform
    variable scoping as in Section 3.5.
end procedure
```

Figure 6. The algorithm to transform a software helper threading loop candidate to a DOALL loop.

which will be illustrated later in detail through an example.

If any checking reveals that the loop in the main thread has completed execution, the helper thread will stop its work immediately. Figure 5(c) shows the transformed code. The loop *t* in Figure 5(c) is marked as a DOALL loop and will be later parallelized with the existing automatic parallelization framework.

3.5. Variable Scoping

For the parallel loop *t* in Figure 5(c), the compiler scopes the variables based on the following rules:

- All arrays and address-taken scalars are shared.
- All non-address-taken scalars (including structure members) are private.
- All private scalars upward-exposed to the beginning of loop *t* are first private.
- All private scalars downward-exposed to the end of loop *t* are both last private and first private. The purpose is to copy out correct value in case that the scalar assignment statement does not execute at runtime.

For any downward exposed variables, the runtime library and outlining code generation have been modified to copy out the downward exposed variables in the main thread since all the original computation is done in the main thread. Figure 6 shows the compiler algorithm to transform a helper threading loop candidate to a DOALL loop.

3.6. Examples

Figure 7(a) shows an example, whose trip counts cannot be computed at compile time. We also assume that the compiler is not able to guarantee that $q \rightarrow data$ and $p \rightarrow next$ access different memory locations at compile time. If profile feedback data is available, the compiler will compute the trip count and branch probabilities based on profile data. Otherwise, the compiler chooses default values for unknown trip counts and branch probabilities as in the Section 3.3. Figure 7(b) shows the two-version parallelization transformation. The b_1 is the potential benefit for helper threading and o_1 is the parallelization overhead. Both are compile-time constants. Therefore, at compile time, the branch will be resolved. Figure 7(c) shows program slicing and variable renaming. Note that the variable *tmp_p* is used to copy the original *p* value. Figure 7(d) shows the added checking codes to end the helper thread earlier, if the helper thread runs behind the main thread. The variable *tmp_c* is used to count the number of iterations in the helper thread. The variable *check_c*, which is a compile-time constant, specifies the number of iterations to check whether the main thread has finished or not. Note that all back edges in the helper thread loop or its inner loops are checked. This is necessary in case that the innermost loop is never or rarely got executed.

4. Runtime Support for Helper Threading

In Figure 5(c), the compiler creates a parallel loop *t* which will spawn the main thread and the helper thread at runtime. Helper threading shares the same runtime as automatic/explicit parallelization. For each helper threading loop, runtime creates one POSIX thread to represent the helper thread. This POSIX thread will be reused as the helper thread for subsequent helper threading loops. Since synchronization may unnecessarily slow down the main thread, if a helper thread runs behind, we do not want them to be synchronized at the end of *parallel for* loop *t* (in Figure 5(c)).

Currently, some data (like loop bounds, first private data and shared data, etc.) are passed from the serial portion

IEEE COMPUTER SOCIETY

```
while (p) {
 if (p→data == 0) {
  q = p→data;
  while (q) {
   q→data = c;
   q = q→next;
 } }
 p = p→next;
}
```
(a)

```
if (b1 >= o1) {
 while (p) {
   q→data = c;
   q = q→next;
  } }
  p = p→next;
} }
else {
 while (p) {
  if (p→data == 0) {
   q = p→data;
   while (q) {
    q→data = c;
    q = q→next;
  } }
  p = p→next;
} }
```
(b)

```
if (b1 >= o1) {
 for (t = 0; t <= 1; t++) {
  if (t == 0) {
   while (p) {
    if (p→data == 0) {
     q = p→data;
     while (q) {
      q→data = c;
      q = q→next;
    } }
    p = p→next;
  } }
  else {
   tmp_p = p;
   while (tmp_p) {
    if (tmp_p→data == 0) {
     q = tmp_p→data;
     while (q) {
      prefetch (&(q→data));
      q = q→next;
    } }
    tmp_p = tmp_p→next;
} } } }
else {
 while (p) {
  p→data = c;
  p = p→next;
} }
```
(c)

```
if (b1 >= o1) {
 is_mt_done = FALSE;
 #pragma omp parallel for
 for (t = 0; t <= 1; t++) {
  if (t == 0) {
   while (p) {
    if (p→data == 0) {
     q = p→data;
     while (q) {
      q→data = c;
      q = q→next;
    } }
    p = p→next;
   }
   is_mt_done = TRUE;
  }
  else {
   tmp_p = p;
   if (is_mt_done == FALSE) {
    tmp_c = 0;
    while (tmp_p) {
     if (tmp_p→data == 0) {
      q = tmp_p→data;
      while (q) {
       prefetch (&(q→data));
       q = q→next;
       if (tmp_c >= check_c) {
        if (is_mt_done == TRUE) {
         goto next;
       } }
       tmp_c = tmp_c + 1;
     } }
     tmp_p = tmp_p→next;
     if (tmp_c >= check_c) {
      if (is_mt_done == TRUE) {
       goto next;
      } }
      tmp_c = tmp_c + 1;
    } }
    next:
} } }
else {
 while (p) {
  p→data = c;
  p = p→next;
} }
```
(d)

Figure 7. A code generation example.

```
procedure ht_main_thread_no_end_sync
        (void *data)
 LOCK
 if (prev_main_data == NULL) then
  prev_main_data = data
 else if (prev_main_data
    ≠ prev_helper_data) then
  free (prev_main_data)
  prev_main_data = data
 else
  prev_main_data = data
 end if
 UNLOCK
end procedure
```
(a)

```
function ht_helper_thread_no_end_sync
        (void *data)
 LOCK
 if (prev_helper_data ≠ NULL) then
  free (prev_helper_data)
 end if
 if (prev_main_data ≠ data) then
  prev_helper_data = NULL
  should_continue = FALSE
 else
  prev_helper_data = data
  should_continue = TRUE
 end if
 UNLOCK
 return should_continue
end function
```
(b)

Figure 8. Action taken by the main thread and the helper thread to free shared parallel data.

of the main thread to the runtime library, and then to the outlined routine, which will be executed by both the main thread and the helper thread. Such data, which we call *shared parallel data*, will be allocated on the heap through *malloc* routine. The runtime system must find a way to free such space to avoid potential our-of-memory issues.

The main thread will access every piece of shared parallel data. However, the helper thread may not, since either it may be suspended, or it runs far behind so skipping some helper threading loops. Also, for every piece of shared data, the main thread will access it first before the helper thread accesses it, since the main thread *activates* the helper thread.

Figures 8(a) and (b) show the action taken by the main thread and the helper thread, respectively, to free shared parallel data. The function parameter is the address of the shared parallel data for the current helper threading loop. The functions are called at the beginning of the main thread and the helper thread inside the runtime library, respectively, before delivering control to the outlined routine. The global variables *prev_main_data* and *prev_helper_data* are used to record the previously accessed shared parallel data by the main thread and the helper thread, respectively, both of which have an initial *NULL* value. If the future accessed shared parallel data by the helper thread are not the one currently accessed by the main thread, the helper thread should not continue the stale helper threading loop, which is indicated by the return value *should_continue*. Since both functions access the shared data, to avoid race condition, the same LOCK/UNLOCK pair is placed in the beginning and the end of both functions.

5. Experimental Results

In this section, we present experimental results with helper threading on the SPEC CPU2000 suite [22]. Our experiments were done on the UltraSPARC IV+ processor as

IEEE COMPUTER SOCIETY

described in Section 2. All the techniques described in Sections 3 and 4 have been implemented in a prototype based on SUN's production compiler. We compare our helper threading performance with the best serial performance obtained using the peak compiler flags in SUN's SPEC submission [22]. Currently, our runtime system does not automatically detect and bind the main thread and the helper thread to the same chip. We do this manually using the environment variable SUNW_MP_PROCBIND [24].

Table 1 shows the distribution of loops in each program including the number of loops accepted and rejected for helper threading. The "total" is the number of loops examined by the procedure in Figure 3. Note that if an outer loop is considered a candidate and profitable, its inner loops are not counted in the "total". The "bad shape" column shows the number of loops rejected because of multiple loop exits or unknown control flow. Note that to outline a loop, we need a single-entry single-exit region. The "calls" column shows the number of loops rejected because they contains calls with side effects. The "bad comp" column shows the number of loops rejected because the execution of address computation or branch resolution codes may raise an exception. The "no save" column shows the number of loops rejected because the compiler was not able to find any effective prefetch candidate. The "other opt" column shows the number of loops rejected because the compiler determines that outlining it would hurt some other optimization [1]. The "overrun" column shows the number of loops rejected because the compiler determines that the helper thread may run too far ahead of the main thread and overwrite the shared cache. The "no benefit" column shows the number of loops rejected because the compiler considers them not profitable for helper threading as discussed in Section 3.3. The "2 ver" column shows the number of loops for which two versions with a runtime check are generated as described in Section 3.3. The "1 ver" column shows the number of loops for which two version code is not generated. The "%load_stalls" column shows the percentage of execution time spent stalled for L2 load cache misses (when helper threading is not used). This is measured using processor performance counters. Note that this counter does not include store stall time which could also be reduced by helper thread prefetching. Note that in Table 1, the only two-versioned loops are in `swim`. This is because the peak SPEC submission, to which we compare our work, used profile feedback for all benchmarks except `swim`.

Figure 9 shows the speedup achieved with helper threading (HT). The "GM" shows the geometric mean speedup of all CPU2000fp or CPU2000int benchmarks, respectively. Three benchmarks, `equake`, `lucas`, and `mcf`, achieve a speedup of over 1.1, with a maximum speedup of 1.22

[1]Currently, this is added to avoid certain regressions. This is an area we will continue to improve.

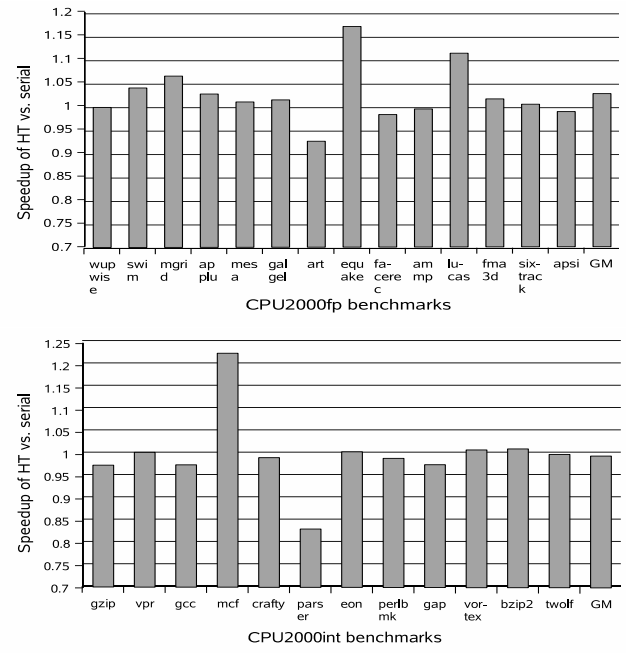

Figure 9. Comparison between helper threading vs. serial performance.

for `mcf`. `fma3d` suffers L2 cache misses significantly (Table 1), while our helper threading does not improve performance much. We have work to do in the future in both interleaved prefetching and helper threading for this benchmark. `Art` and `parser` run slower with helper threading than without it. This is because memory accesses in the regions selected for helper threading for these two benchmarks are mostly cache resident and the overhead of helper threading becomes significantly greater than the corresponding gain. We have focused on helper threading without cache miss profiling information, in order to increase its acceptance and ease of use. However, we do plan to study and utilize cache miss profiling to avoid such regressions.

To further understand the performance difference, we measured the number of L2 cache misses using performance counters. Figure 10 shows the reduction in L2 cache misses due to helper threading for the main thread. For all benchmarks, helper threading is able to either maintain or reduce the number of L2 cache misses. For `equake`, L2 cache misses are reduced by over 94%. For `crafty` and `twolf`, there is a large reduction in L2 cache misses but it does not translate into performance gain because these programs have a very low L2 cache miss rate to begin with (see Table 1). For `gzip`, although L2 cache misses are reduced by 25%, performance with helper threading rather degrades by 2%. This is due to compiler phase ordering problems and is an area we are trying to improve currently.

1089-795X/05 $20.00 © 2005 IEEE

IEEE COMPUTER SOCIETY

Table 1. Distribution of loops for helper threading.

benchmarks	total	bad shape	calls	bad comp	no save	other opt	overrun	no benefit	2 ver	1 ver	%load_stalls
wupwise	197	0	55	4	4	0	0	132	0	2	0.1%
swim	77	0	19	0	3	0	0	55	55	0	0.8%
mgrid	54	1	19	0	2	0	0	28	0	4	6.8%
applu	85	8	12	0	1	0	0	64	0	0	4.3%
mesa	1452	57	25	8	6	0	0	1354	0	2	2.0%
galgel	452	9	38	5	19	0	0	369	0	12	0.8%
art	85	3	10	13	6	0	0	52	0	1	0.3%
equake	85	0	14	3	39	0	0	27	0	2	9.3%
facerec	236	1	38	1	8	3	0	184	0	1	1.4%
ammp	423	209	14	4	4	0	0	179	0	13	9.8%
lucas	83	9	10	0	1	0	0	52	0	11	4.3%
fma3d	5530	84	62	5	242	0	0	5128	0	9	17.8%
sixtrack	2358	127	130	15	125	0	0	1954	0	7	0.5%
apsi	389	10	57	7	1	0	0	309	0	5	3.0%
gzip	301	50	38	0	41	0	0	165	0	7	0.2%
vpr	623	175	157	2	24	0	0	250	0	15	5.1%
gcc	3449	1340	563	0	178	0	0	1352	0	16	4.0%
mcf	66	17	3	0	3	1	0	34	0	8	30.2%
crafty	416	76	164	0	18	0	1	155	0	2	0.08%
parser	791	310	193	0	85	0	0	195	0	8	3.1%
eon	572	73	33	7	0	0	0	459	0	0	0.0%
perlbmk	1195	426	128	0	22	0	0	619	0	0	2.1%
gap	2744	647	446	0	34	7	1	1603	0	6	8.0%
vortex	230	129	15	0	6	1	0	79	0	0	5.6%
bzip2	192	35	62	0	32	0	0	53	0	10	6.0%
twolf	1047	164	175	6	185	1	0	481	0	35	0.07%

Due to space limitations, we briefly summarize the results of some other interesting experiments we have done.

- It is always profitable to not synchronize at the end of the parallel *for* in Figure 5(c). The performance difference with and without such synchronization is within 3% for all benchmarks except *mcf*, where the performance difference is 16%.
- Interleaved prefetching in the main thread remains important even with helper threading. This is because the helper thread only brings the data to the L2 cache and software prefetching in the main thread often does a better job of bringing data from the L2 cache to the prefetch cache than hardware prefetching [7]. For example, prefetching in the main thread can achieve a speedup of 1.8 compared to no prefetching in the main thread for `swim`.
- Though we have attempted helper threading without cache miss profiling, we use conventional block count profiling for all benchmarks except `swim`. Without this information, the performance of `mgrid`, `apsi` and `gcc` degraded by 23%, 22% and 8%, respectively, This was due to excessive two-version code generation. In the future, we plan to study how to improve our profitability test and *judiciously* generate two-version code, for cases where no profile feedback information is available.
- The gains described in Figure 9 are over very aggressive SPEC peak options. We have also obtained the gains due to helper threading with more typical compiler options and where interleaved prefetching is not used. When helper threading is added to this scenario, prefetching is done entirely by the helper thread. Figure 11 shows the results, which clearly demonstrate that the helper thread brings data successfully into the shared L2 cache and significantly accelerating the main thread. The geometric mean improves by a factor of 1.30X for CPU2000fp where several benchmarks suffer large L2 cache miss penalties. Most CPU2000int benchmarks do not suffer such penalties and here our helper threading causes minimal degradation.

6. Related Work

Helper threading is not new. Kim *et al.* present their helper threading experience on a hyperthreaded Pentium(TM) processor [8]. They rely on cache miss profile data for helper threading while we use regular edge profiling or a two-version scheme if profiling data is not available. We have targeted a dual-core design where the helper thread does not contend for pipeline resources with the main thread. Our work does not use any special hardware support for helper threading other than a shared L2 cache. We have presented a detailed and systematic code generation scheme for helper threading. Kim *et al.* present a source-to-source transformation and simulation framework for helper threading [9, 10]. They use a preprocessor while we directly im-

IEEE COMPUTER SOCIETY

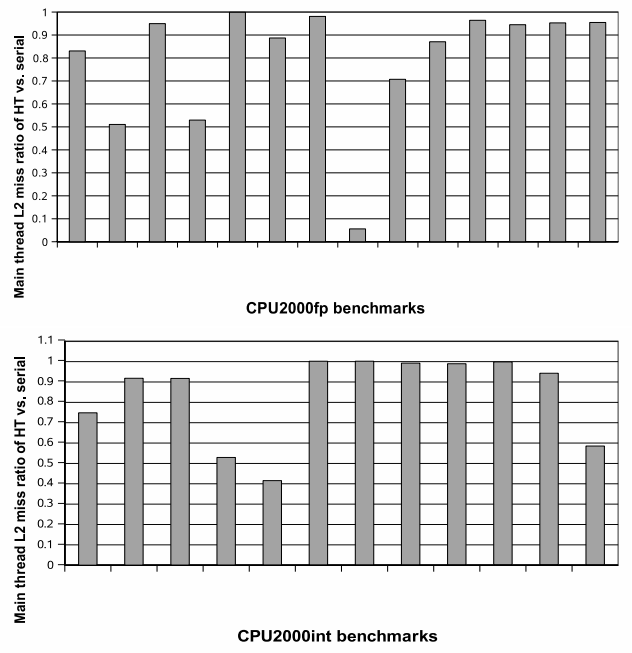

Figure 10. Ratio of L2 cache misses for data of the main thread between helper threading vs. serial performance.

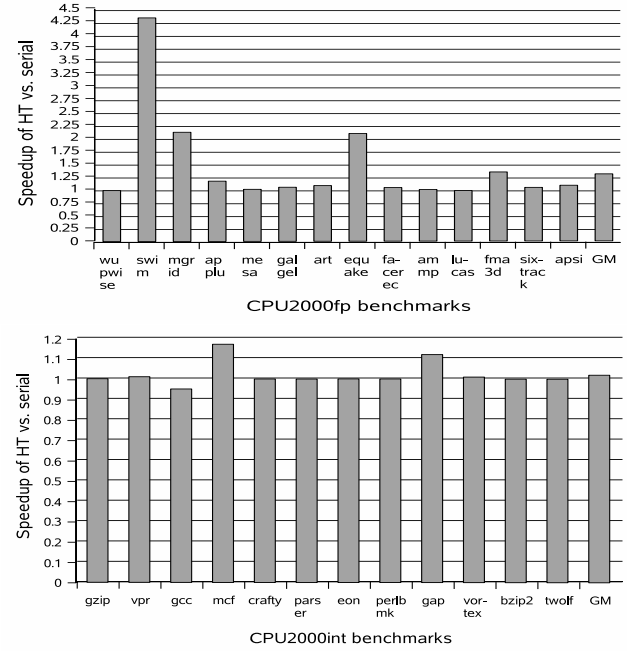

Figure 11. Speedup of helper threading vs. serial performance without interleaved prefetching.

plement helper threading inside the compiler. They also require special hardware support.

A number of researchers have tried to construct helper threads by post-processing binaries. Liao *et al.* use a binary rewriter to generate helper threads [12]. Collins *et al.* use speculative pre-computation for prefetching [5]. Luk describes the use of helper threading in the simultaneously multithreaded machines [13]. Both [5] and [13] require special hardware support. Brown *et al.* evaluate helper threading on chip multiprocessors through simulation and also propose several architectural enhancement for better helper threading on chip multiprocessors [2], while we conduct our experiments on real hardware. There have also been attempts to construct helper threads or dynamically perform prefetching at runtime. Moshovos *et al.* tried this approach by using a special hardware slicing processor [17].

Prefetching for linked-list data structures and general prefetching in a single thread are also not new. Roth and Sohi use jump pointers to compute prefetching targets [21]. Choi *et al.* show a prefetch engine to perform multi-chain prefetching, a technique to exploit inter-chain memory parallelism [4]. Wu uses value profiling to predict the regularity in irregular streams [28]. Mowry *et al.* show how prefetching can help performance for regular predictable memory streams in both a uniprocessor and a multiprocessor system [18, 19]. Tullsen and Eggers describe techniques for prefetching effectively in a multiprocessor system [25].

7. Conclusion

In this paper, we have presented a compiler framework to perform helper threading. We have shown techniques for selecting candidate loops, deciding whether candidate loops are profitable, and generating code that is resilent to operating system scheduling. Our method operates without cache miss profile data and without special hardware support. We have implemented our framework in a production compiler and evaluated it on a dual-core processor. In our experiments using the SPEC CPU2000 benchmark suite, helper threading improved performance for codes suffering large L2 cache miss penalties without substantially degrading the performance of the others. The maximum gain observed was 22%.

The following aspects of helper threading merit further study. Easy-to-use cache miss profile information could lead to increased performance gains and reduced regressions. Special hardware support for helper threading would reduce overheads. Given the emergence of processors with a large number of threads, the possibility of having more than one helper thread for a main thread can be considered. New heuristics might help improve prefetching in the helper thread. For example, the indexed array access prefetching

IEEE COMPUTER SOCIETY

technique [7] can be applied in the helper thread by default because helper threads can afford to do more work in support of prefetching. Finally, improvements in the runtime library, e.g. automatic processor binding, can make helper threading more effective.

References

[1] A.-R. Adl-Tabatabai, R. L. Hudson, M. J. Serrano, and S. Subramoney. Prefetch injection based on hardware monitoring and object metadata. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 267–276, June 2004.

[2] J. Brown, H. Wang, G. Chrysos, P. Wang, and J. Shen. Speculative precomputation on chip multiprocessors. In *The 6th Workshop on Multithreaded Execution, Architecture and Compilation*, November 2002.

[3] B. Cahoon and K. McKinley. Data flow analysis for software prefetching linked data structures in java. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, 2001.

[4] S. Choi, N. Kohout, S. Pamnani, D. Kim, and D. Yeung. A general framework for prefetch scheduling in linked data structures and its application to multi-chain prefetching. *ACM Transactions on Computer Systems*, 22(2):214–280, May 2004.

[5] J. Collins, H. Wang, D. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. Shen. Speculative precomputation: Long-range prefetching of delinquent loads. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 14–25, June 2001.

[6] D. Greenley. The ultrasparc iv+ processor. In *Proceedings of the Fall Processor Forum*, San Jose, CA, October 2004.

[7] S. Kalogeropulos, M. Rajagopalan, V. Rao, Y. Song, and P. Tirumalai. Processor aware anticipatory prefetching in loops. In *Proceedings of the 10th International Conference on High-Performance Computer Architecture*, pages 106–117, February 2004.

[8] D. Kim, J. P. Shen, S. S. Liao, P. H. Wang, J. del Cuvillo, X. Tian, X. Zou, H. Wang, D. Yeung, and M. Girkar. Physical experimentation with prefetching helper threads on intel's hyper-threaded processors. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 27–38, Palo Alto, CA, March 2004.

[9] D. Kim and D. Yeung. Design and evaluation of compiler algorithms for pre-execution. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 159–170, San Jose, CA, 2002.

[10] D. Kim and D. Yeung. A study of source-level compiler algorithms for automatic construction of pre-execution code. *ACM Transactions on Computer Systems*, 22(3):326–379, August 2004.

[11] G. Lauterbach. Ultrasparc iii - a scalable high clock rate sparc processor. In *Proceedings of the Microprocessor Forum*, San Jose, CA, October 1997.

[12] S. S. Liao, P. Wang, H. Wang, G. Hoflehner, D. Lavery, and H. Shen. Post-pass binary adaptation for software-based speculative precomputation. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 117–128, Berlin, Germany, June 2002.

[13] C.-K. Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 40–51, 2001.

[14] C.-K. Luk and T. Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of the 7th International Conference on Architecture Support for Programming Languages and Operating Systems*, October 1996.

[15] C.-K. Luk and T. Mowry. Automatic compiler-inserted prefetching for pointer-based applications. *IEEE Transactions on Computers*, 48(2):134–141, February 1999.

[16] C.-K. Luk and T. Mowry. Architectural and compiler support for effective instruction prefetching: A cooperative approash. *ACM Transactions on Computer Systems*, 19(1), February 2001.

[17] A. Moshovos, D. N. Pnevmatikatos, and A. Baniasadi. Slide-processors: An implementation of operation-based prediction. In *Proceedings of the International Conference on Supercomputing*, Sorrento, Italy, 2001.

[18] T. Mowry. Tolerating latency in multiprocessors through compiler-inserted prefetching. *ACM Transactions on Computer Systems*, February 1998.

[19] T. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the 5th International Conference on Architecture Support for Programming Languages and Operating Systems*, October 1992.

[20] OpenMP ARB, OpenMP Specification Version 2.5. *http://www.openmp.org*.

[21] A. Roth and G. Sohi. Jump-pointer prefetching for linked data structures. In *Proceedings of the 26th International Symposium on Computer Architecture*, May 1999.

[22] Standard Performance Evaluation Corporation, The SPEC CPU2000 benchmark suite. *http://www.specbench.org*.

[23] Sun Microsystems, Inc., Sun Studio 9: Performance Analyzer. *http://docs.sun.com/source/817-6696/AdvancedTopics.html*.

[24] Sun Microsystems, Inc., Sun Studio 9: OpenMP API User's Guide. *http://docs.sun.com/source/817-6703/3_Compiling.html*.

[25] D. M. Tullsen and S. J. Eggers. Effective cache prefetching on bus-based multiprocessors. *ACM Transactions on Computer Systems*, 13(1), February 1995.

[26] S. Vanderwiel. Data prefetch mechanisms. *ACM Computing Surveys*, 32(2), June 2000.

[27] D. Weaver and E. T. Germond. *The SPARC Architecture Manual Version 9*. PARC International, Inc., Prentice-Hall, Englewood Cliffs, NJ.

[28] Y. Wu. Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching. In *Proceedings of the International Conference on Programming Language Design and Implementation*, June 2002.

1089-795X/05 $20.00 © 2005 **IEEE**

IEEE COMPUTER SOCIETY