CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 0000; **00**:1–18
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

Compiler Multiversioning for Automatic Task Granularity Control

Peter Thoman Herbert Jordan Thomas Fahringer

University of Innsbruck, Institute of Computer Science
Technikerstrasse 21a, 6020 Innsbruck, Austria
{petert,herbert,tf}@dps.uibk.ac.at

SUMMARY

Task parallelism is a programming technique that has been shown to be applicable in a wide variety of problem domains. A central parameter that needs to be controlled to ensure efficient execution of task-parallel programs is the granularity of tasks. When they are too coarse-grained, scalability and load balance suffer, while very fine-grained tasks introduce execution overheads.
We present a combined compiler and runtime approach that enables automatic granularity control. Starting from recursive, task parallel programs, our compiler generates multiple versions of each task, increasing granularity by task unrolling. Subsequently, we apply a parallelism-aware optimizing transformation to remove superfluous task synchronization primitives in all generated versions. A runtime system then selects among these task versions of varying granularity by locally tracking task demand.
Benchmarking on a set of task parallel programs using a work-stealing scheduler demonstrates that our approach is generally effective. For fine-grained tasks, we can achieve reductions in execution time exceeding a factor of 6, compared to state-of-the-art implementations. Additionally, we evaluate the impact of two crucial algorithmic parameters, the number of generated code versions and the task queue length, on the performance of our method.

Received ...

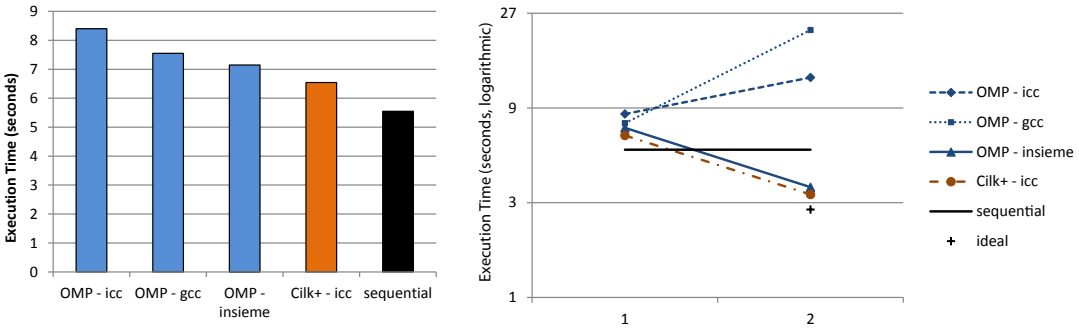
KEY WORDS: Compiler; Runtime System; Parallel Computing; Task Parallelism; Multiversioning; Recursion

1. INTRODUCTION

Task-based parallelism is one of the most fundamental parallel abstractions in common use today [1]. While relatively easy to implement and use, achieving good efficiency and scalability with task parallelism can be challenging. A central feature of every task-based parallel program that significantly affects both efficiency and scalability is *task granularity* [2]. The *granularity* of tasks is defined by the length of the execution time of a single task between interactions with the runtime system, such as spawning new tasks.

Very fine-grained, short-running tasks lead to a loss in efficiency compared to sequential execution due to the runtime overhead associated with generating and launching a task, as well as synchronizing its completion with other tasks in the system. On the other hand, coarse-grained, long-running tasks minimize overhead, but are hard to schedule effectively and may therefore fail to scale well on large parallel systems. Previous work in this area has focused mostly on runtime systems or user-controlled cutoffs to manage granularity (see Section 5). Conversely, we propose an approach that combines a multiversioning compiler with a runtime system which adaptively selects from the generated versions. Our goal is to maximize efficiency by increasing task granularity – and thus decreasing overheads – without negatively affecting load balance or scalability.

We implemented our method for OpenMP [3] tasks within the Insieme compiler and runtime system [4], but the idea is equally applicable to any other task parallel language. Our concrete contributions are the following:



(a) Single-threaded

(b) Scaling from 1 to 2 threads

Figure 1. Initial Experiments, N-Queens $N = 13$.

- A compile-time multiversioning transformation that generates a set of task implementations of increasing granularity by recursive *task unrolling*. This transformation is applicable to both simple recursion and N-ary mutual recursion.
- A compiler transformation removing superfluous synchronization statements in unrolled recursive task parallel programs.
- A runtime heuristic for the dynamic adaptation of granularity based on the concept of *task demand*, which automatically choses the code version to execute at each task spawning point.
- Evaluation and analysis of the performance of our method on a number of well-known task parallel benchmarks. We compare with other OpenMP implementations, our own implementation without the multiversioning optimization and Cilk [5] versions which represent the state of the art in fine-grained task parallelism.

This paper improves upon and extends work previously presented by the authors [6], formalizing the description of the compiler transformations used and evaluating the impact of various compile-time and runtime algorithmic parameters on the performance of the presented method. Furthermore, the behaviour of our runtime task selection heuristic is investigated in detail.

The remainder of this paper is structured as follows. In Section 2 we provide some initial results that motivated our work. We then describe our method in detail in Section 3 and evaluate its performance as well as the impact of various compile-time parameters in Section 4, followed by an overview of related work in Section 5. Finally, Section 6 summarizes and concludes our findings.

2. MOTIVATION

In this section we present some initial benchmark results that motivate our multiversioning method. Figure 1(a) shows single-threaded execution times measured for the Barcelona OpenMP Tasks Suite (BOTS) [7] N-Queens benchmark with $N = 13$. For details on the hardware, compiler versions and programs used refer to Section 4.

The lowest execution time amongst the OpenMP versions is achieved by our compiler and runtime system (Insieme), however, this time is still 28% higher than purely sequential execution. Even the Cilk version, while more efficient than any OpenMP implementation, is 19% slower than the sequential version. Our multiversioning method is designed to address this inefficiency. Throughout this paper, when we refer to *inefficient* execution, we mean execution with a single thread which takes longer than executing purely sequential code, or, for multiple threads, longer than starting from the sequential time and assuming perfect scaling (that is a speedup of N with N cores and threads in hardware).

Note that the OpenMP runtime systems of ICC [8] and GCC [9] perform special case handling when only a single worker thread is used. This is visible in Figure 1(b), which shows their performance degrading when switching from one to two threads. Further experiments in Section 4

DOI: 10.1002/cpe

confirm this behavior, with scaling starting after some initial performance degradation when activating multi-threaded execution. The OpenMP version compiled with Insieme and the Cilk version do not suffer from this issue, however they still induce a relative overhead of about 20% compared to ideal linear scaling from the sequential version. We identified the following potential causes for this inefficiency:

1. Task generation overhead. This includes generating a task structure, populating it with values and enqueuing it. The overhead of this operation depends on the amount of private data each task requires.
2. Synchronization primitive overhead (e.g. `taskwait`). At the very least, this involves keeping track of all the subtasks launched by each task, and signaling when they are complete.
3. Task library calls. The runtime methods required for tasking are generally implemented in a separate library, and the overhead for their invocation is incurred even if they perform no actual work.
4. Non-inlineable, indirect program function calls. Since the program function implementing a given task needs to be called by the tasking library, a pointer to it is usually passed to the library function. Even if the runtime library decides to directly execute the call, this prevents the benefits – improved instruction scheduling and a reduction in overhead – associated with inlining.

Issues 1 and 2 can be mitigated by a pure runtime approach, e.g. the runtime library can dynamically decide whether to generate a full task structure or directly call the task function. This method is usually referred to as lazy task creation [10]. However, the basic overhead of library function calls (issue 3) and the fact that indirectly called functions in the original program can not be inlined (issue 4) can not be changed at runtime and need to be handled at compile time. This limitation of pure runtime systems motivates our compiler-aided multiversioning approach.

All four potential causes for inefficient execution identified above are directly related to and influenced by the granularity of tasks. The more often individual tasks are generated and synchronized, the higher the impact of the associated overheads on execution time. However, simply increasing the granularity of all tasks is not a solution: such an approach will lead to load imbalance, and it increases the probability of workers idling.

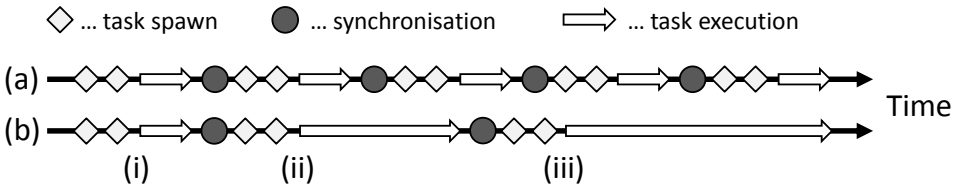


Figure 2. Timelines for default task execution and execution with variable granularity.

Therefore, our goal, as illustrated in Figure 2, is the generation of different implementations for each task. The upper timeline (a) shows the default execution of a single worker thread in a task-parallel program. All tasks have the same granularity and execution time. The lower timeline (b) depicts our ideal goal, involving dynamic selection from a set of implementations at each task spawning point. Early on, at point (i), a fine-grained task is generated so that the desired degree of parallelism is achieved quickly. At later points (ii) and (iii) the system is saturated and therefore the task granularity is gradually increased, reducing the inherent overhead caused by any interactions with the runtime library.

3. METHOD

Figure 3 illustrates the major components of our proposed method. Starting from an OpenMP program with parallel tasks, our compiler generates an application in which multiple different

Prepared using cpeauth.cls
DOI: 10.1002/cpe

implementation versions of each task are encoded. During execution of the program, whenever a specific task is invoked, our runtime system selects and launches a version of this task. The static compiler transformations utilized during the multiversioning process are detailed in Section 3.1, while 3.2 describes the scheduling heuristic employed in the runtime system.

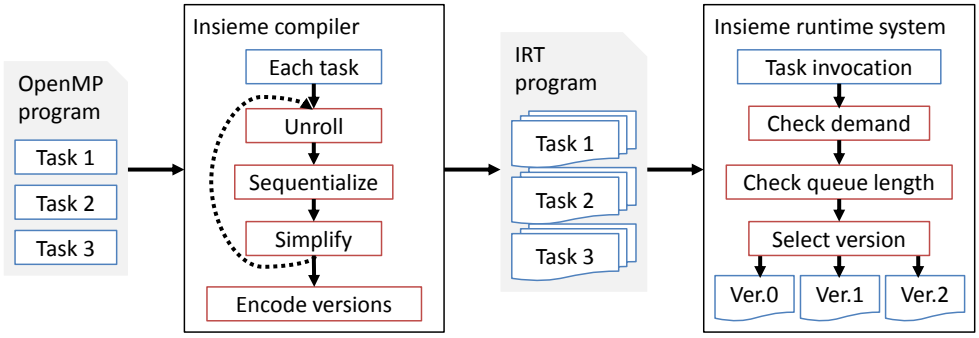


Figure 3. Overview of our Method.

3.1. Compile-time Multiversioning

During compilation our goal is to generate multiple versions of each parallel task, with varying granularity. As depicted in Figure 3 this involves a three step process, which may be applied multiple times to further increase the task size. The individual steps are as follows:

1. **Task unrolling**. Replaces each nested task invocation site with a direct call to the task function, which is subsequently inlined. This can be thought of as a context and parallelism-aware recursive function inlining step. The name *task unrolling* is adapted from Rugina's usage of *recursion unrolling* [11].
2. **Sequentialization**. This step focuses on identifying which synchronization primitives – if any – were rendered superfluous by the partial elimination of parallel task invocations due to task unrolling, and removing them. It is described in more detail below.
3. **Simplification**. Unrolling and sequentialization may have generated redundant or sub-optimal code, particularly after multiple iterations. In the simplification step, we apply a number of well known sequential optimizing compiler transformations to ease further processing of the code and simplify unnecessarily complex structures that may have been introduced during the previous processing steps. The transformations applied include inlining of very small function calls, constant folding, copy propagation, algebraic simplification, strength reduction and unused code elimination [12].

The number of generated versions depends on the granularity of the initial tasks and the largest granularity desired. The versions are generated and encoded into the output program in the following order.

1. **Original**. The original version from the input program.
2. N **times unrolled versions**. Starting from $N = 1$. In these versions, only partial sequentialization is performed. Outer task spawning points are removed, but the innermost spawning location is kept. This process is illustrated in detail in a code example in Figure 5, described below.
3. **Fully sequentialized version**. In this version all task spawning points are removed and replaced with plain function calls.

Figure 4 illustrates the result of generating 3 versions for a mutually recursive task set consisting of two functions $F1$ and $F2$. The original program (Listing 1) thus has four task spawning locations, two calling the task spawning wrapper code of F1 (As) and two the corresponding wrapper of F2

DOI: 10.1002/cpe

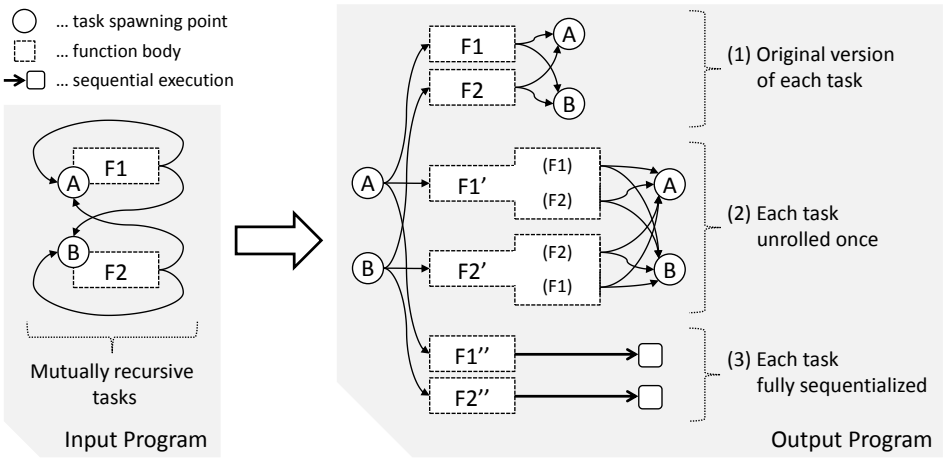


Figure 4. Version Generation and Control Flow.

(Bs). To improve the clarity of the illustration, these task spawning wrappers have been replicated in the transformed version, however they are still all referring to the same task.

```
1  f1(n) = {
2      if(n == 0) return 0;
3      a = spawn f1(n-1)+1;                  // A = potentially spawning a call to f1
4      b = spawn f2(n-1)+2;                  // B = potentially spawning a call to f2
5      merge_all();
6      return a + b;
7  }
8  f2(n) = {
9      if(n == 0) return 0;
10     a = spawn f1(n-1)+10;                 // A = potentially spawning a call to f1
11     b = spawn f2(n-1)+20;                 // B = potentially spawning a call to f2
12     merge_all();
13     return a + b;
14 }
```

Listing 1: Version Generation Example Code.

Version (1) is identical to the original program, except that at each spawning point there is now a choice between 3 distinct implementations of each function. In version (2), consisting of $F1'$ and $F2'$, each recursive task invocation was unrolled once, forming tasks of increased granularity. Clearly, if this version is used, more work is performed between individual task invocations and interactions with the runtime library. Finally, version (3), comprising $F1''$ and $F2''$, is fully sequentialized. Once this version is invoked, no further parallel tasks will be spawned on this branch of the recursive descent.

3.1.1. Code Example Figure 5 illustrates the effect of the steps taken during compilation to generate a task version that has been unrolled once. A pseudo-code formulation is used for reasons of clarity and size. It is C-like, but without the need for explicit type specification, and with two additional keywords: `spawn` implies the generation of a new parallel task (corresponding to `#pragma omp task untied` in OpenMP), while `merge_all` waits for the completion of all launched subtasks (equivalent to `#pragma omp taskwait`).

In (a), the original input code is shown. Moving on to (b), first-level task invocations are removed and replaced with in-place calls of the associated functions. Context-sensitive inlining of these calls results in (c). Finally, redundant applications of the `merge_all` operation are removed and arithmetic simplification is applied. The final generated code for this version is listed in (d). This process can be repeated N times to generate increasingly larger task sizes.

DOI: 10.1002/cpe

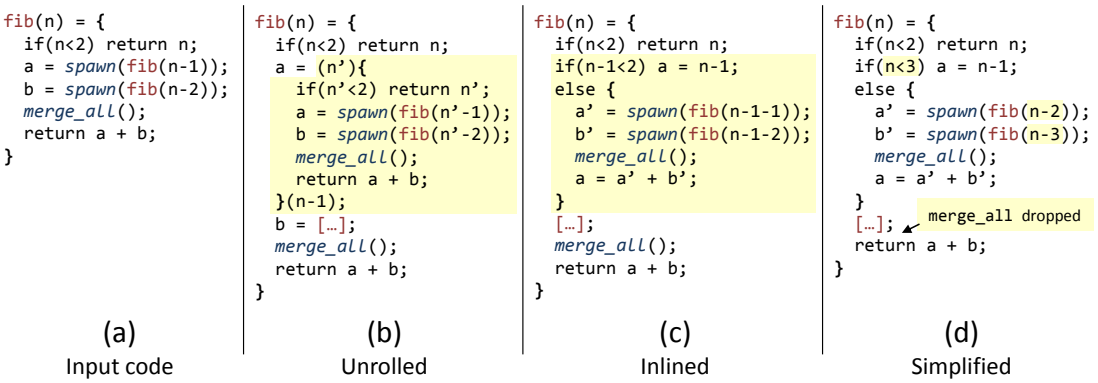


Figure 5. Example task transformation - Fibonacci - Version generation.

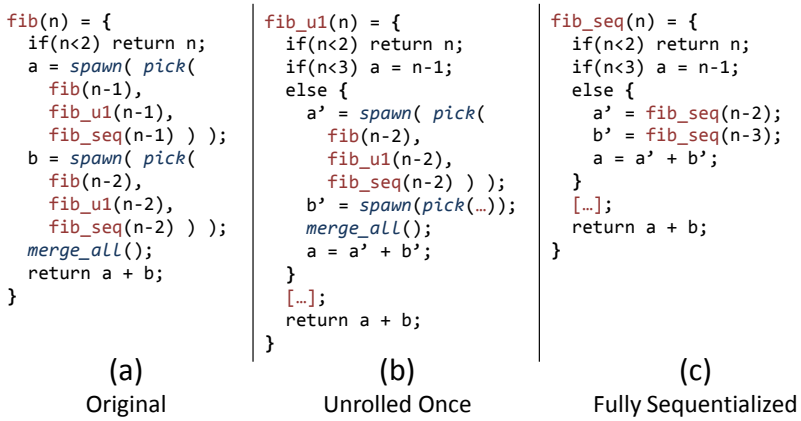


Figure 6. Example task transformation - Fibonacci - Generated versions.

After all the versions are generated, each version needs to be modified to enable runtime selection. Figure 6 contains the final code for the original version with task selection (a), the unrolled version as discussed previously (b) and a fully sequentialized version (c).

The `pick` keyword in Figure 6 implies a possible choice between semantically equivalent versions, which is deferred to the runtime system. That is, in terms of program semantics, $\texttt{spawn}(\texttt{pick}(a, b, c)) \equiv \texttt{spawn}(a) \equiv \texttt{spawn}(b) \equiv \texttt{spawn}(c)$. The intention is for a, b and c to differ in non-functional parameters, such as execution time, memory usage, or – as in this case – degree of parallel execution. This choice is included at the spawning points of the original version, as well as all unrolled versions. In the fully sequentialized version, the spawning point is removed and replaced with a direct recursive call to the sequentialized function.

3.1.2. Partial Sequentialization In most parallel programs there will be some superfluous synchronization statements after task unrolling. Since the execution has been partially sequentialized, instructions that wait for the completion of a task that was unrolled are no longer necessary and should be removed. The transformation eliminating unnecessary synchronization acts as detailed in Algorithm 1 on a task version T, effectively removing all `merge_all` operations for which there is no possibility of any task being spawned between them and a previous `merge_all`.

3.1.3. Synchronization Elimination Example As an example, Algorithm 1 is applied to the task version generated in Figure 5 (c). The full code for this stage in the version generation is given in Listing 2, and we will refer to the code statements by their line number, as well as the labels added for spawn and merge operations. Computing the set M as per the algorithm for this example yields $M = \{m_a, m_b, m_1\}$.

DOI: 10.1002/cpe

Algorithm 1 Superfluous Synchronization Elimination Algorithm.

T input/output task version

1: Determine the set M of all `merge_all` invocations in T.
2: **for all** `merge_all` $m \in M$ **do**
3: Compute the set of all static execution paths F from the entry point of T to m.
4: **for all** paths $f \in F$ **do**
5: Reverse f and remove the first entry.
6: **end for**
7: **if** $\forall f \in F : f$ encounters no `spawn` before a `merge_all` **then**
8: Remove m from T.
9: **end if**
10: **end for**

```
1  fib(n) = {
2    if(n<2) return n;
3    if(n-1<2) a = n-1;
4    else {
5      a' = spawn(fib(n-1-1));                    // s_a1
6      b' = spawn(fib(n-1-2));                    // s_a2
7      merge_all();                               // m_a
8      a = a' + b';
9    }
10   if(n-2<2) b = n-2;
11   else {
12     a' = spawn(fib(n-2-1));                    // s_b1
13     b' = spawn(fib(n-2-2));                    // s_b2
14     merge_all();                               // m_b
15     b = a' + b';
16   }
17   merge_all();                                 // m_1
18   return a + b;
19 }
```

Listing 2: Synchronization Elimination Example.

For m_a, the set $F = \{(1, 5, 6, 7)\}$. Reversing the single included path and removing the first entry results in $F = \{(6, 5, 1)\}$. The statement at line 6 is s_{a2}, a `spawn` operation, thus m_a is kept. For m_b the situation is similar, and a `spawn` operation is encountered immediately on the reversed paths, but the paths are slightly more complex.
Finally, consider m_1. In this case, the initial set of paths is given by F_A below. Reversing each path and removing the first entry results in F_B.

$$
\begin{aligned}
F_A = \{ \quad & f'_0 = (1, 5, 6, 7, 8, 12, 13, 14, 15, 17), \\
& f'_1 = (1, 3, 12, 13, 14, 15, 17), \\
& f'_2 = (1, 5, 6, 7, 8, 10, 17), \\
& f'_3 = (1, 3, 10, 17) \quad \}
\end{aligned}
\qquad
\begin{aligned}
F_B = \{ \quad & f_0 = (15, 14, 13, 12, 8, 7, 6, 5, 1), \\
& f_1 = (15, 14, 13, 12, 3, 1), \\
& f_2 = (10, 8, 7, 6, 5, 1), \\
& f_3 = (10, 3, 1) \quad \}
\end{aligned}
$$

On the path f_0, the `merge_all` operation m_b is encountered at 14, before any `spawn`. On f_1, the situation is the same. On f_2, the `merge_all` operation m_a is encountered at 7, again before any `spawn`. No `spawn` operation is contained in f_3. Thus, the condition holds for all paths and m_1 can safely be eliminated.

Note that in this particular example a much simpler algorithm, such as removing the first `merge_all` operation postdominating the spawns which have been replaced, would also be sufficient. However, it is possible to encounter cases where there is no postdominating `merge_all`

DOI: 10.1002/cpe

call – e.g. when distinct merges are performed in different execution paths through the task, but there is at least one merge in each path. These cases are also handled by our method.

3.2. Runtime Version Selection

The previous section outlined how multiple versions with different granularities and trade-offs are generated in the compiler. This provides the runtime system with an opportunity of making a version choice every time a task is spawned. Making the wrong choice can result in reduced efficiency, or, at worst, greatly diminish parallelism – e.g. in case a fully sequentialized version is chosen too early. We considered the following design goals and observations when developing our version selection method:

- At the start of the program, the original (most fine-grained) version of the tasks should be used, since the parallelism available in the system is not yet fully leveraged and load-balancing is a priority.
- The impact of conservative behavior – i.e. using more fine-grained tasks – causes more gradual performance degradation than using tasks that are too coarse grained, potentially leading to some worker threads idling.
- The decision procedure needs to be simple, causing only little overhead, otherwise it could negate any benefits from multiversioning.
- The decision making process should be distributed – no new synchronization points between worker threads should be introduced to facilitate version selection.

Taking these points into account led to the development of a distributed version selection heuristic based on two parameters which are tracked for each individual worker thread. The first parameter is *task demand*, which keeps track of other worker's unfulfilled attempts to steal tasks from the local worker. The second parameter is the *queue length* of each worker, which indicates how many tasks it currently has available for execution or stealing.

Task demand is tracked in a surprisingly simple, but effective, manner. The demand is stored locally as an integer which starts at a positive value equal to the maximum task queue length. Whenever a task is generated by a worker thread, it reduces its own task demand value by 1. When a worker k_1 attempts to steal from another worker k_2 which has no tasks available, then the task demand value of k_2 is reset to the maximum task queue length.

Algorithm 2 Task Version Selection Algorithm.

`queue_length`	current queue length
`task_demand`	current task demand
`NUM_VERSIONS`	number of versions generated for current task
`MAX_QUEUE`	maximum queue length (fixed)

output:

0	$\Leftrightarrow$	original task
$N = 1 \ldots$ `num_versions` $- 2$	$\Leftrightarrow$	unrolled N times
`num_versions` $- 1$	$\Leftrightarrow$	fully sequentialized

```
1: version = NUM_VERSIONS − ⌈(task_demand/MAX_QUEUE) ∗ NUM_VERSIONS⌉
2: if version >= NUM_VERSIONS − 1 then
3:     if queue_length == MAX_QUEUE then
4:         return NUM_VERSIONS − 1                          ▷ choose sequential
5:     end if
6:     return NUM_VERSIONS − 2                              ▷ most coarse grained
7: end if
8: return version                                           ▷ gradually adapt granularity
```

Our version selection procedure is listed in Algorithm 2. In conjunction with the demand tracking outlined above, it has the following desirable properties:

DOI: 10.1002/cpe

- Evaluating the selection function only takes a few dozen cycles, assuming that all the required values are cached.
- The way in which task demand is reset to the initial value if any work item stealing operation fails, but is only reduced gradually during normal execution, mirrors the earlier observation about the negative performance impact of wrong granularity selection. It makes the expensive case of idle workers unlikely by reacting very strongly to failed stealing attempts.
- Selecting the fully sequentialized version is a step that should only be taken after careful consideration, since it will prevent any further parallelism from being generated on this branch of the recursive descent. Therefore, the heuristic only takes this step if there has been no demand for additional tasks over a large number of spawn points *and* the queue is full.

The choice of the fixed `MAX_QUEUE` and `NUM_VERSIONS` parameters has an impact on the effectiveness of this approach, which is investigated in Section 4.4. For the comparison to other systems and primary experiments in Section 4, `MAX_QUEUE` was set to 32 and `NUM_VERSIONS` was set to 4.

3.2.1. Task Version Selection Example Let us assume for this example that 4 code versions were generated for a given work item corresponding to a task, that is $\texttt{NUM\_VERSIONS} = 4$. Given the mapping in Algorithm 2, this means that version 0 is the original code, in version 1 recursive task invocations have been unrolled once, in version 2 they have been unrolled twice and version 3 is fully sequentialized.

Now, assume two workers k_1 and k_2, and $\texttt{MAX\_QUEUE} = 4$ (a very low value chosen for illustrative purposes). Both workers start with a `task_demand` $t^d_{k_1} = t^d_{k_2} = 4$, and will execute the simple Fibonacci code sample shown previously (Figures 5 and 6), with k_1 starting the outermost task execution. At the start of the program, $\texttt{version} = 4 - \lceil (4/4) * 4 \rceil = 0$, which means that the initial code version (with the smallest granularity) is chosen. This results in the spawning of two new work items, decrementing $t^d_{k_1}$ by 1 each, resulting in $t^d_{k_1} = 2$. At this point, k_2 may or may not steal work items from k_1 – it does not change the further execution unless the queue in k_1 becomes empty. If that happens, $t^d_{k_1}$ gets reset to 4. We assume for this example that this does not occur.

When k_1 selects a follow-up task version, the selection algorithm will evaluate to $\texttt{version} = 4 - \lceil (2/4) * 4 \rceil = 2$. Thus, the 2 times unrolled version is selected, which generates 8 new work items. This will fill up the queue and set $t^d_{k_1} = 0$. At this point, as long as the queue remains full, $\texttt{version} >= \texttt{num\_versions} - 1 \;\&\&\; \texttt{queue\_length} == \texttt{MAX\_QUEUE}$ will evaluate to true, and the next task will be executed with full sequentialization. As soon as k_1's queue loses an element, either because it is stolen by k_2 or executed by k_1 itself, it will fall back to choosing the unrolled but not fully sequentialized code version 2, immediately refilling the queue. Thus, unless a stealing attempt fails later on, this particular program will complete using primarily the highly efficient fully sequential code version, with some interspersed partially unrolled versions.

4. EVALUATION

In this section we will evaluate the performance impact of our optimization. Subsection 4.1 details our measurement methodology and the experimental setup used. We will perform an in-depth evaluation of two programs in Subsection 4.2, proceed with an overview of the results of a number of other codes (4.3), investigate the impact of algorithmic parameters in 4.4 and conclude with an analysis of the runtime behaviour of our version selection algorithm in 4.5.

4.1. Experimental Setup

For our experiments we used an Intel-based parallel system, incorporating 4 Xeon E7-4870 processors, each comprising 10 physical cores (20 hardware threads) and 3 levels of cache. Table I summarizes the configuration of this system.

Prepared using cpeauth.cls
DOI: 10.1002/cpe

Table I. Hardware and software platform for experimental evaluation.

Sockets/ Cores	Cache L1d/i	L2	L3	Software OS	Kernel	GCC	ICC	Insieme
4/40	32K/32K	256K	30M	CentOS 6.3	2.6.32	4.6.3	12.1	g4614502

When running experiments using a subset of cores, all involved threads were bound to individual physical cores such that the resources of one chip are fully utilized before involving an additional processor. All experimental runs were repeated five times, and the median runtime is reported.

While the most important comparison for our evaluation is between our compiler with and without our multiversioning method, we also included the results obtained by other platforms to provide a reference for comparison. Table I includes the exact version number of the compilers used in these comparisons. ICC was used as the backend compiler for the Insieme source to source infrastructure, and its built-in Cilk Plus support was employed to compile Cilk programs. The optimization flag "`-O3`" was enabled for all calls to GCC and ICC. The source code for Insieme can be obtained from the authors.

4.2. A Detailed Evaluation

4.2.1. N-Queens The first program we will evaluate is the N-Queens benchmark included in BOTS [7]. Each task in N-Queens spawns 0 to N child tasks, and the depth of its task invocation trees varies from 1 to N, while not following any simple pattern. The size of individual tasks is relatively small.

Figure 7 illustrates the performance of N-Queens using a variety of compilers and implementations. Four OpenMP versions are shown: GCC, ICC and Insieme with ("taskopt") and without ("insieme") task optimization. Additionally, we included the results of a Cilk version and a fully sequential version without any parallel language primitives. The execution time is presented in a log-log plot to improve readability. An efficiency plot is also provided, which compares the execution times of the parallel versions against ideal scaling from the sequential version.

In terms of OpenMP results, it is clear that the task granularity in this benchmark is too small to be handled effectively by GCC's GOMP implementation. ICC shows the same behavior that was already partially observed in Section 2 – execution time increases when going from a single-threaded to a multi-threaded setup. However, starting from two threads performance scales relatively well up to 40. Since both of these OpenMP implementations seem ill-equipped to handle very fine-grained tasking well, we also included a Cilk version, which has previously been shown to provide

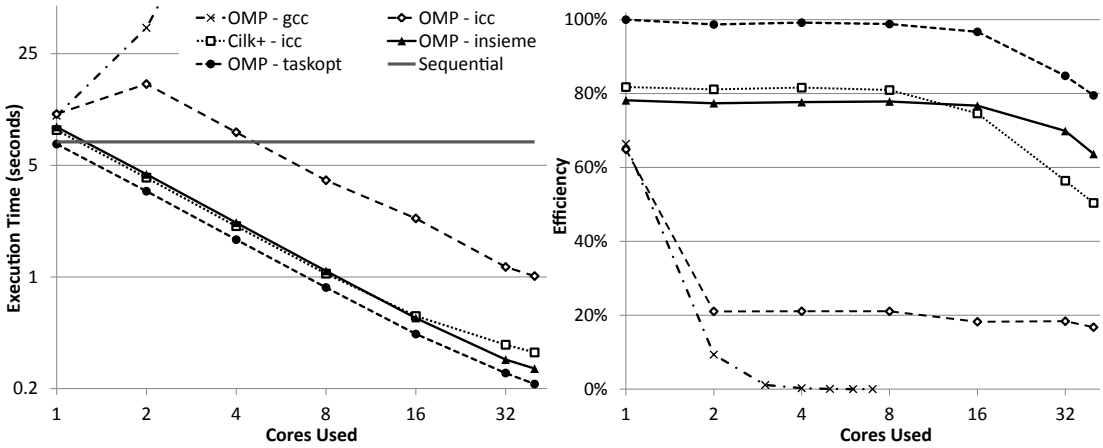


Figure 7. N-Queens benchmark results, $N = 13$.

DOI: 10.1002/cpe

better scaling for fine-grained tasks [13]. Indeed, this implementation performs better in the single-threaded case and scales more smoothly to multiple cores than the GCC and ICC OpenMP versions.

Using Insieme to compile the OpenMP input program results in performance that is comparable to Cilk for up to 16 cores, and scales slightly better beyond this amount. However, a comparison with the fully sequential version indicates that even the Insieme OpenMP version and the Cilk version lose around 20% of performance to overheads incurred due to parallelization. When our task optimization – that is, multiversioning in the compiler and adaptive work item implementation version selection at runtime, as presented in the previous sections – is activated, this overhead is effectively avoided. Even more importantly, this significant reduction in overhead is achieved without negatively affecting the scalability of the program. Performance compared to our implementation without task optimization is improved by 22% to 28% across all measured core counts, with a 25% increase at the full 40 cores.

Compared to the fully sequential version, our approach achieves an efficiency above 99% up to 8 cores, 97% at 16 cores, 85% with 32 cores and 80% at 40 cores. The total runtime of our implementation at higher core counts goes below 0.3 seconds. Note that the drop-off in efficiency primarily occurs at 16 cores and above. This is due to the problem size N=13 causing each initial task to spawn 13 sub-tasks, which means that up to 13 cores can be supplied with work during the first ”generation” of tasks. When more cores are used, a larger number of second-generation (and beyond) tasks need to be distributed.

Using the full system (40 cores), our implementation with task optimization improves N-Queens performance by 56% compared to the best competing implementation (Cilk).

4.2.2. Fibonacci For a second in-depth evaluation, we chose the BOTS Fibonacci program. This is very similar to the code example provided in Section 3 (Figure 5). As a test case, its most interesting features compared to N-Queens are a significantly different shape of the task invocation tree and the extremely small size of individual tasks. In Fibonacci, each task only creates zero to two sub-tasks, however the maximum depth of the task invocation tree is much larger. Additionally, the depth of the task chains follows an easily predictable pattern, unlike N-Queens.

Note that this is obviously an inefficient method of generating the Fibonacci numbers which would not be used in a production code. However, its properties make it an interesting case study for the overhead of task-parallel systems.

The performance results achieved in Fibonacci by the set of implementations included in our comparison are illustrated in Figure 8, again adjusted to a log-log scale to make them easier to interpret.

The issues with small tasks experienced by the OpenMP implementation in GCC are exacerbated in this case, due to the extremely fine task granularity of the Fibonacci program. We stopped the execution of this version after 15000 seconds in the case of 5 or more threads, since these results

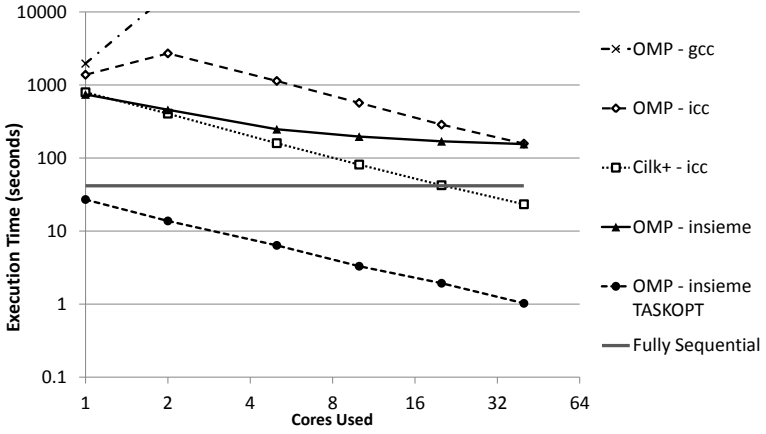


Figure 8. Fibonacci benchmark results, $N = 48$.

DOI: 10.1002/cpe

have no meaningful impact on the comparison. ICC's OpenMP implementation acts similarly to before, with special handling of the single-threaded case and good scaling after the initial parallelism overhead. Cilk is about twice as fast as ICC's OpenMP version in the single-threaded case, and scales well from that point.

As with N-Queens, our Insieme OpenMP implementation without task optimization starts out similarly to Cilk with one and two cores used. However, with a larger number of cores, our scaling behavior suffers. This is due to the very fine-grained nature of the tasks, and the fact that generating a stealable work item induces more overhead in our implementation than it does for Cilk.

However, the most significant result is the performance of the purely sequential version compared to any other existing implementation. Even the most efficient parallel implementations are slowed down by a *factor of 20* when comparing their single-threaded execution time to the fully sequential program. The only existing system that manages to improve on the sequential result at all is Cilk, and it requires 20 cores to do so.

With such small tasks, and therefore relatively large parallelism overheads, it is reasonable to expect that our multiversioning scheme as introduced in this paper will have a large impact on performance. As Figure 8 shows, both absolute performance and scalability are greatly improved. With a single thread, runtime is reduced by a *factor of 26* compared to our implementation without multiversioning and adaptive granularity adjustment. Interestingly, the single-threaded execution time using our system is even lower than that of the fully sequential program, due to recursion unrolling, which is not performed by ICC or GCC. This mirrors earlier results in the field of sequential optimization [11], and shows that for very fine-grained tasks, even sequential function call overheads have a relevant impact.

When using all 40 cores of the system, our new approach improves upon the best existing solution (Cilk) by a factor of 23.

4.3. Further Benchmarks

Table II summarizes our benchmark results. It includes measurements for the N-Queens and fib benchmarks presented above, as well as a number of additional programs.

Sort Is the *sort* benchmark included in BOTS.

Strassen Also from BOTS, matrix multiplication using the Strassen algorithm.

Table II. Benchmark Results.

cores	1	2	5	10	20	40	cores	1	2	5	10	20	40
Queens, $N = 13$ - seq: 7.42													**Fib,** $N = 48$ - seq: 31.09
gcc	10.23	36.29	148.28	308.16	545.22	725.98	gcc	1960.35	17093.63	>15000	>15000	>15000	>15000
icc	10.49	16.04	6.45	3.81	1.60	0.91	icc	1379.84	2705.65	1135.29	569.15	286.41	157.70
ins	8.69	4.35	1.74	0.87	0.46	0.27	ins	742.40	456.95	247.91	196.59	169.50	155.29
opt	6.79	3.41	1.48	0.69	0.36	0.21	opt	27.06	13.77	6.37	3.30	1.93	1.03
imp	27.92%	27.52%	17.78%	26.64%	25.35%	24.91%	imp	26.43×	32.17×	37.90×	58.15×	86.69×	150.36×
Sort, $N = 2^{27}$ - seq: 21.51													**Strassen,** $N = 8192$ - seq: 158.15
gcc	21.98	11.80	7.20	17.17	29.43	42.29	gcc	159.74	92.45	39.20	22.10	15.36	19.94
icc	23.87	12.36	5.04	2.80	1.85	1.56	icc	164.43	89.94	39.12	21.81	15.69	19.27
ins	22.94	12.00	4.90	2.71	1.93	1.53	ins	168.84	85.97	37.51	21.98	12.94	8.72
opt	20.81	11.18	4.61	2.52	1.72	1.41	opt	154.27	79.80	35.46	19.81	12.03	8.11
imp	5.61%	5.47%	6.43%	7.47%	7.88%	8.11%	imp	3.54%	7.72%	5.77%	10.08%	7.55%	7.52%
Stencil, $N = 2048$ - seq: 18.90													**Floorplan,** `input.20` - seq: 17.86
gcc	46.82	62.09	138.51	398.05	576.83	840.61	gcc	27.36	31.04	133.30	352.94	514.51	759.20
icc	30.17	24.65	15.63	14.64	13.84	12.04	icc	*	*	*	*	*	*
ins	32.49	18.48	9.27	6.31	7.50	9.67	ins	23.53	12.48	5.05	2.53	1.72	1.58
opt	24.96	13.84	6.66	4.26	5.15	7.54	opt	17.20	9.51	4.12	2.09	1.43	1.24
imp	20.87%	33.49%	39.17%	47.97%	45.50%	28.29%	imp	36.76%	31.25%	22.62%	21.06%	20.52%	27.68%
FFT, $N = 2^{29}$ - seq: 184.78													**QAP,** `chr18a` - seq: 237.28
gcc	222.27	132.66	95.88	276.81	420.00	482.07	gcc	488.97	931.43	7471.11	>15000	>15000	>15000
icc	189.73	112.13	55.95	37.44	22.64	16.03	icc	785.36	2539.80	823.00	319.87	179.58	114.93
ins	187.36	104.85	51.39	36.46	21.01	16.96	ins	578.57	294.13	112.80	78.65	70.97	60.71
opt	183.97	100.02	49.66	35.08	19.07	12.03	opt	231.62	110.76	40.24	21.88	15.18	9.90
imp	1.84%	4.84%	3.48%	3.93%	10.16%	33.21%	imp	2.11×	2.66×	2.80×	3.59×	4.68×	6.13×

DOI: 10.1002/cpe

Stencil A task based 2D stencil computation using the cache-oblivious algorithm presented by Frigo and Strumpen [14]. We included this benchmark to represent an important category of cache-oblivious divide-and-conquer algorithms.

Floorplan The BOTS *floorplan* benchmark. For this application, the binary generated by ICC 12.1 repeatably caused a segmentation fault within ICC's OpenMP library, regardless of the number of threads used. Therefore we are unable to present ICC results for this benchmark.

FFT A parallel fast fourier transform included in BOTS.

QAP A branch and bound solver for quadratic assignment problems.

For every benchmark, the table contains five rows. The results achieved using the GCC and ICC OpenMP implementations are listed in the "gcc" and "icc" rows, respectively. The "ins" row contains the results of our Insieme compiler and runtime without the task multiversioning optimization presented in this paper, while it is enabled for the measurements listed in the "opt" row. Finally, the values in the "imp" row represent the relative improvement achieved using adaptive granularity control, compared to the best result among the other three versions. The columns labeled 1 to 40 correspond to the number of cores used for the computation. All times are given in seconds, and the improvement is provided in percent, except in the case of the Fibonacci and QAP benchmarks where improvement factors are listed instead of very large percentages.

As a frame of reference, the purely sequential time for each benchmark compiled with ICC is provided in each header ("seq"). Note that this time falls between the Insieme time without optimization and the optimized version in most cases, except in the stencil test. Here, the restructuring performed by our compiler prevents some of the low-level sequential optimizations performed by ICC. However, our optimized version executed with one thread is still closer to the sequential performance than any other parallel implementation.

A general trend visible throughout all the benchmark results is the relationship between default task granularity, scaling in GCC and the degree of improvement possible using adaptive task multiversioning and selection. The Fibonacci and QAP benchmarks have the most fine grained tasks, and consequently the worst scaling in GCC and the largest improvement with our optimization. On the other end of the spectrum, the FFT, strassen and sort benchmarks feature built-in cutoff values that inherently control task granularity by preventing very small tasks from being generated, resulting in more modest, but still significant, performance improvements with multiversioning. Floorplan, stencil and N-queens fall in between these extremes.

One interesting behavioral pattern which merits some explanation occurs in FFT. Our multiversioning implementation does not result in any significant improvement up to 10 cores, however at 40 cores the measured improvement is 33%. This is due to the FFT benchmark consisting of two separate phases: coefficient calculation and FFT computation. These phases exhibit distinct scaling behaviour, and one of them is affected more significantly by adaptive granularity optimization than the other. Thus, with a larger number of cores, the phase with bad scaling starts to take up a larger portion of the execution time, and the effect of multiversioning on overall performance increases.

4.4. Impact of Algorithmic Parameters

The compiler transformations, version generation and runtime version selection which constitute our proposed method are influenced by two parameters: `MAX_QUEUE` and `NUM_VERSIONS`. As described in Section 3, the former determines the maximum task queue length – before immediate execution sets in – of our runtime system, while the latter specifies the number of different code versions generated by the compiler for each task. As each subsequent task version is unrolled once more, `NUM_VERSIONS` also implicitly caps the maximum task unrolling depth. Both of these parameters influence the runtime task selection heuristic, as presented in Algorithm 2.

In the results provided above, `NUM_VERSIONS` was set to 4 and `MAX_QUEUE` was set to 32. While not optimal in all cases, these values were empirically determined to provide stable results across a wide range of programs and hardware configurations. To gauge the impact of varying these

Prepared using cpeauth.cls
DOI: 10.1002/cpe

parameters, we evaluated program performance across all plausible settings for both. We will now demonstrate the results of this investigation on the N-Queens program.

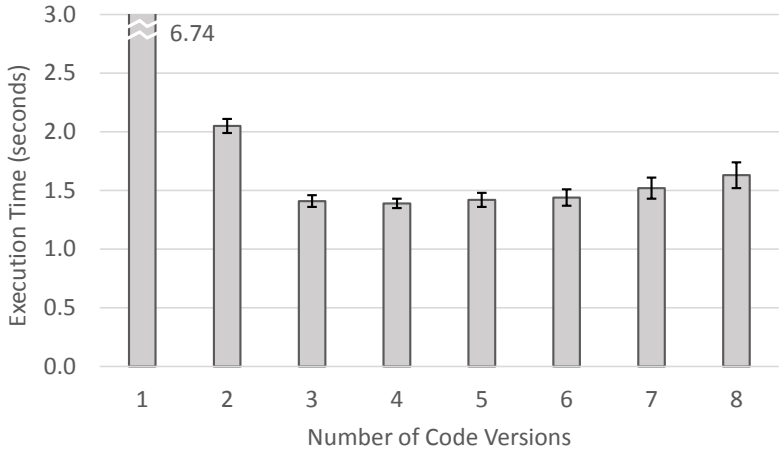


Figure 9. N-Queens results with varying degree of multiversioning, $N = 14$.

4.4.1. Number of Code Versions Figure 9 illustrates the performance of the N-Queens benchmark using a fixed setting of 40 threads and varying number of task code versions, and thus maximum task unrolling depth. We only show the results for one thread count configuration, as, for this parameter, the number of threads does not meaningfully affect the result.

Using only a single task version is equivalent to running the original, unmodified input program. With two task versions, the options for each task are either the original fine-grained task or fully sequentializing the execution. Even the availability of such a binary choice already improves performance significantly, reducing execution time by a factor of 3.3. Note that the performance achieved with this option is quite close to the state-of-the art task-parallel Cilk performance.

Leveraging the unique flexibility of our approach by adding a third code version, the runtime system has the options of either choosing the original task, a more coarse-grained but still parallel version of the original task, or fully sequentializing the current branch of the recursion. This improves performance further, with an additional 32% reduction in execution time compared to 2 code versions. Adding one more intermediate version, for a total of 4, insignificantly improves the result (by less than 2%), while going beyond 5 versions gradually decreases performance. This decrease can be attributed to a large number of code versions – all of which are intermittently active during the program execution – leading to worse utilization of the CPU instruction cache, while not meaningfully improving the scheduling flexibility of our algorithm.

An additional factor to consider when choosing `NUM_VERSIONS` is the impact of this setting on compile time, both in our source-to-source compiler as well as the backend compiler. Since the generated code size for a given task unrolling factor N is exponential in the number of (mutually) recursive task invocations, compile times may start increasing significantly when going beyond a maximum unrolling factor of 2 – which is achieved with a total of 4 generated versions. For large `NUM_VERSIONS` this can be an obstacle – we have observed compile times of 35 minutes for 8 versions, even on our relatively small benchmark codes. However, as the performance results above indicate, such a large number of versions is not beneficial in practice, so this issue does not occur.

4.4.2. Task Queue Length Unlike the number of code versions, the impact of the choice of maximum task queue length differs greatly depending on the number of threads used to execute a program. Therefore, Figure 10 includes the results for 40, 10 and 1 thread(s) of execution, each normalized to their respective execution time using the default queue length of 32 elements in order to enable an effective visual comparison.

DOI: 10.1002/cpe

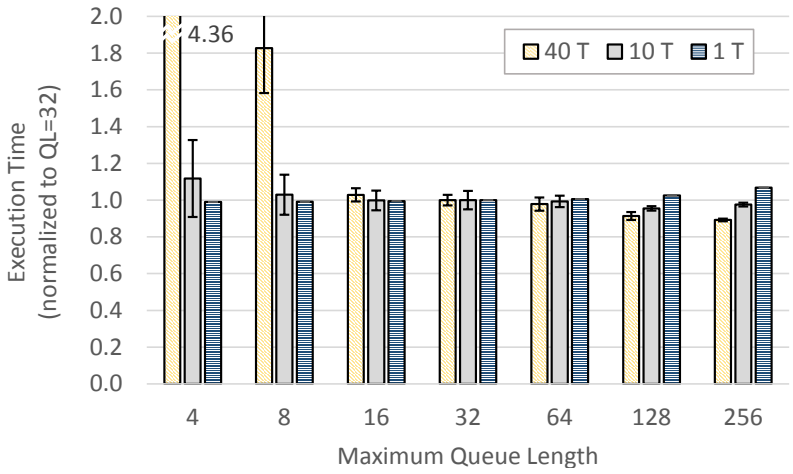


Figure 10. N-Queens results with varying task queue length, $N = 14$.

For the single-threaded case, larger queue sizes gradually increase execution times, reaching a total slowdown of 8% for a queue length of 256. This is a direct consequence of the runtime version selection as described in Algorithm 2: a longer queue will lead to fully sequentialized task versions – which are obviously ideal in the single-threaded case – being chosen later during program execution, compared to a shorter queue.

In both multi-threaded cases, we observe a significant loss in performance when the queue length is decreased, particularly if it drops below the number of worker threads. The most extreme impact can be observed for 40-way parallel execution using a queue length of just 4 tasks, which results in a slowdown by a factor of 4.36 compared to the default queue length of 32. These results are caused by load imbalance. If a thread very quickly switches to fully sequential execution of a large branch of the overall task tree after only filling a short queue, other threads might run out of work and be unable to acquire any new tasks by stealing after all queues have been emptied. Larger queue length are more beneficial with many threads, and with 40 threads a queue length of 256 actually performs best in this benchmark. For 10 threads, ideal performance is reached with a queue length of 128, after which the same effect observed for a single thread starts to outweigh any improvement in load balance.

Note that the variance across all test runs is illustrated by whiskers in the plot, and that there is negligible variance in the single-threaded results. For multi-threaded runs, the variance increases significantly with short queues, as the exact order and targets of initial task stealing operations influence the degree of load imbalance during the entire program run.

In conclusion, the task queue length parameter should be adapted per-system to fit the amount of available hardware parallelism, and choosing a queue size larger than the optimum is generally less likely to cause abrupt performance degradation. Automatically choosing a suitable queue length for some input code and hardware setup based on static analysis is a subject of future work.

4.5. Version Selection Analysis

As a final step in investigating the performance properties of our method, we will look at the behavior of Algorithm 2 during runtime. For this purpose, we instrumented the version selection procedure in order to collect statistical data characterizing the mixture of selected code versions. Table III and Figure 11 illustrate the result of this analysis.

The columns in Table III show, in order: the number of *Workers* used, the *Total* amount of version selection events, the average amount of version selection events per worker (*PerW*), the number of times each version was chosen (*V0s* to *V3s*), and the number of restart events encountered (*Rst*), as well as the relative percentage of these 5 values in the total number of events. A restart event is defined as a point in the execution where previously a version numbered > 0 was selected, and

Prepared using cpeauth.cls
DOI: 10.1002/cpe

Table III. Version Selection in N-Queens.

Workers	Total	PerW	V0s	V1s	V2s	V3s	Rst	%V0s	%V1s	%V2s	%V3s	%Rst
1	1547	1547	2	2	2	1541	0	0.12	0.12	0.12	99.61	0.00
2	3432	1716	14	10	10	3398	11	0.40	0.29	0.29	99.00	0.32
5	6890	1378	64	38	38	6750	58	0.92	0.55	0.55	97.96	0.84
10	12610	1261	438	168	194	11810	427	3.47	1.33	1.53	93.65	3.38
20	39286	1964	4425	1156	1412	32293	4404	11.26	2.94	3.59	82.19	11.21
40	81133	2028	23578	2860	3400	51295	23537	29.06	3.52	4.19	63.22	29.01

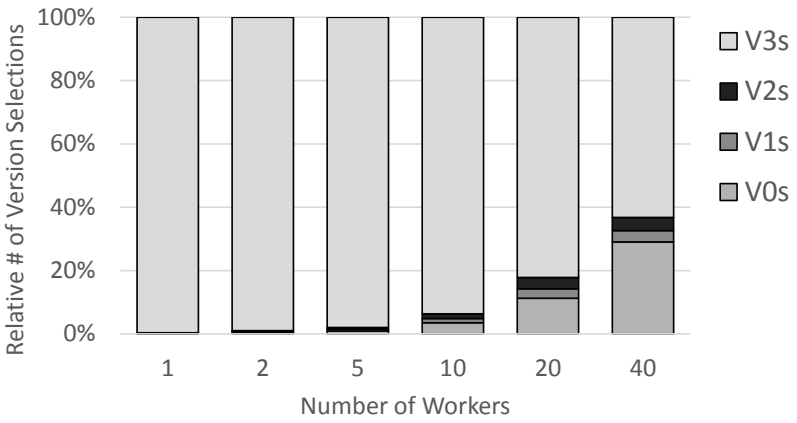


Figure 11. N-Queens version choice with varying number of worker threads, $N = 14$.

subsequently version 0 (the most fine-grained version) is selected again due to an unsatisfied task stealing request.

A first observation on this data is that the most coarse-grained version (3) dominates the execution in all cases except for the ones which feature a high degree of parallelism – i.e. with 20 and 40 worker threads. This is a desirable property, as that code version allows for minimal execution time overhead, and should be preferred as long as all worker threads in the system have a sufficient supply of tasks available.

In terms of scalability, we observe that while the total number of selection events grows with the degree of parallelism, the number of selection events per worker remains relatively constant. As such, the overhead in execution time for version selection does not grow significantly with a larger number of active workers.

The number of reset operations is indicative of how the runtime algorithm reacts to failed work stealing attempts. For execution with one thread, there are no failed stealing events, thus there are also no resets, and the most coarse-grained code version is always selected after a short initial warm-up period. This explains why executing the parallel program with a single thread using our system generally matches (or, in some cases, exceeds) the performance of the unmodified sequential program. When the number of threads is increased, the system needs to react to the probability of some of them running out of work, and therefore we see an increasing number of reset events. This in turn also increases the relative number of times code versions smaller than the most coarse grained (i.e. versions 0, 1 and 2) are chosen. In this fashion a dynamic balance between generating sufficient parallel work and executing the program with as little overhead as possible is achieved, which is directly responsible for the good absolute performance and scalability of our method for fine-grained task parallel programs, as observed in Section 4.3.

DOI: 10.1002/cpe

5. RELATED WORK

Much previous work on parallel tasks has focused on runtime systems [15] or scheduling policies [16]. As described in section 2, pure runtime modifications are incapable of dealing with all the causes for inefficiency that our combined compiler and runtime approach covers. Moreover, our proposed multiversioning scheme is orthogonal to scheduling decisions and can be combined with any scheduling policy.

A common approach towards dealing with task granularity issues is having the user provide thresholds or cut-off values [2]. In our work, task granularity is controlled entirely by the compiler and runtime system, without requiring manual programmer support. Duran et al. [17] describe an adaptive cut-off method which does not require manual adjustment, but their pure runtime approach does not offer the performance benefit of full sequentialization in the compiler.

Inlining of recursive functions has been previously performed in sequential program transformation [18], even with the express purpose of improving performance in divide and conquer programs by reducing overheads [11]. However, these works do not deal with parallelism, while our approach focuses primarily on minimizing the overhead incurred by parallel task creation and synchronization.

Some recent publications have used compiler multiversioning in a parallel setting [19][20], but they focused exclusively on loop-based data parallelism. Conversely, our multiversioning approach is designed for task-parallel, recursive programs.

Very recently, Deshpande and Edwards used recursion unrolling to improve opportunities for parallelism in Haskell programs [21]. Unlike our method, they do not use multiversioning or version selection at runtime, and their compiler transformations are designed for the Haskell functional language while we process input programs written in C with OpenMP.

6. CONCLUSION

We have presented a fully automatic, adaptive approach to parallel task granularity control which goes beyond what can be achieved by improving either just a runtime system or focusing only on compilation. By combining a compiler which performs task multiversioning with a runtime system that adaptively selects from these versions, we were able to minimize parallel runtime overhead even for very fine grained tasks. Our method uses a novel combination of compiler transformations to build an optimized set of semantically equivalent task versions which differ in granularity. The availability of this set of implementations in the compiled program in turn enables our runtime heuristic to adjust the amount of tasks generated, while incurring even less overhead than a traditional lazy task creation system with cut-offs.

Evaluating our proposed method across a set of benchmarks has shown that our optimization is widely applicable, and that the magnitude of these improvements is related to the task granularity of the input program. For programs with relatively coarse-grained tasks, execution times are reduced by 5% - 10%, while we can achieve improvements of a factor of 6 or more compared to the best competing implementations in fine-grained test cases. Varying the number of generated code versions indicates that at least 3 separate task granularities should be generated, while going beyond 4 versions gradually decreases performance due to cache pressure. The ideal task queue length for our approach grows with the number of independent hardware threads available.

Benchmark results also demonstrate that our runtime selection heuristic successfully ensures that scalability (up to 40 cores) is not negatively affected by adaptive task granularity adjustment. Crucially, our adaptive granularity control scheme improves performance in all tested benchmarks and for any given number of cores.

ACKNOWLEDGEMENTS

This research was partially funded by the Austrian Research Promotion Agency under contract nr. 834307 (AutoCore) and the chist-era project GEMSCLAIM.

DOI: 10.1002/cpe

REFERENCES

1. Asanovic Kea. The landscape of parallel computing research: A view from berkeley. *Technical Report*, EECS Department, University of California 2006.
2. (ed) DNT, Loidl HW, Hammond K. On the granularity of divide-and-conquer parallelism. *Glasgow Workshop on Functional Programming*, Springer-Verlag, 1995; 8–10.
3. OpenMP Architecture Review Board. OpenMP Specification. Version 3.1 2011. URL `http://www.openmp.org/mp-documents`.
4. Insieme compiler and runtime infrastructure. URL `http://insieme-compiler.org`.
5. Blumofe RD, Joerg CF, Kuszmaul BC, Leiserson CE, Randall KH, Zhou Y. Cilk: an efficient multithreaded runtime system. *Proc. 5th ACM SIGPLAN symp. on Principles and practice of parallel programming*, PPOPP '95, 1995; 207–216.
6. Thoman P, Jordan H, Fahringer T. Adaptive granularity control in task parallel programs using multiversioning. *Euro-Par 2013 - Parallel Processing, 19h International Euro-Par Conference, Aachen, Germany, Proceedings*, Lecture Notes in Computer Science, Springer, 2013.
7. Duran A, Teruel X, Ferrer R, Martorell X, Ayguade E. Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. *Proc. 2009 Int. Conf. on Parallel Processing*, ICPP '09, 2009; 124–131.
8. Intel. Intel c and c++ compilers. http://software.intel.com/en-us/c-compilers/ 2012.
9. Stallman R. Using and porting the gnu compiler collection. *M.I.T. Artificial Intelligence Laboratory* 2001; .
10. Mohr E, Kranz DA, Halstead RH, Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems* 1991; **2**.
11. Rugina R, Rinard MC. Recursion unrolling for divide and conquer programs. *Proc. 13th Int. Workshop on Languages and Compilers for Parallel Computing*, LCPC '00, 2001; 34–48.
12. Bacon DF, Graham SL, Sharp OJ. Compiler transformations for high-performance computing. *ACM Comput. Surv.* Dec 1994; **26**(4):345–420.
13. Olivier S, Prins JF. Comparison of openmp 3.0 and other task parallel frameworks on unbalanced task graphs. *International Journal of Parallel Programming* 2010; **38**(5-6):341–360.
14. Frigo M, Strumpen V. Cache oblivious stencil computations. *Proc. 19th int. conf. on Supercomputing*, ICS '05, 2005; 361–366.
15. Broquedis F, Gautier T, Danjean V. Libkomp, an efficient openmp runtime system for both fork-join and data flow paradigms. *Proc. 8th int. conf. on OpenMP in a Heterogeneous World*, IWOMP'12, 2012; 102–115.
16. Olivier SL, Porterfield AK, Wheeler KB, Spiegel M, Prins JF. Openmp task scheduling strategies for multicore numa systems. *Int. J. High Perform. Comput. Appl.* May 2012; **26**(2):110–124.
17. Duran A, Corbalán J, Ayguadé E. An adaptive cut-off for task parallelism. *Proc. 2008 ACM/IEEE conf. on Supercomputing*, SC '08, 2008; 36:1–36:11.
18. Fitzpatrick S, Clint M, Kilpatrick P. Unfolding recursive function definitions using the paradoxical combinator 1996.
19. Chen X, Long S. Adaptive multi-versioning for openmp parallelization via machine learning. *Proc. 15th Int. Conf. on Parallel and Distributed Systems*, ICPADS '09, 2009; 907–912.
20. Jordan H, Thoman P, Durillo J, Pellegrini S, Gschwandtner P, Fahringer T, Moritsch H. A multi-objective auto-tuning framework for parallel codes. *Proc. 2012 ACM/IEEE Int. Conf. on Supercomputing*, 2012.
21. Deshpande NA, Edwards SA. Statically unrolling recursion to improve opportunities for parallelism. *Technical Report*, Department of Computer Science, Columbia University 2012.

DOI: 10.1002/cpe