Task-parallel Runtime System Optimization Using Static Compiler Analysis

Peter Thoman, Peter Zangerl and Thomas Fahringer
{petert,peterz,tf}@dps.uibk.ac.at
University of Innsbruck

ABSTRACT

Achieving high performance in task-parallel runtime systems, especially with high degrees of parallelism and fine-grained tasks, requires tuning a large variety of behavioral parameters according to program characteristics. In the current state of the art, this tuning is generally performed in one of two ways: either by a group of experts who derive a single setup which achieves good – but not optimal – performance across a wide variety of use cases, or by monitoring a system's behavior at runtime and responding to it. The former approach invariably fails to achieve optimal performance for programs with highly distinct execution patterns, while the latter induces some overhead and cannot affect parameters which need to be fixed at compile time.

In order to mitigate these drawbacks, we propose a set of novel static compiler analyses specifically designed to determine program features which affect the optimal settings for a task-parallel execution environment. These features include the parallel structure of task spawning, the granularity of individual tasks, and an estimate of the stack size required per task. Based on the result of these analyses, various runtime system parameters are then tuned at compile time.

We have implemented this approach in the Insieme compiler and runtime system, and evaluated its effectiveness on a set of 12 task parallel benchmarks running with 1 to 64 hardware threads. Across this entire space of use cases, our implementation achieves a geometric mean performance improvement of 39%.

CCS CONCEPTS

•**Software and its engineering** → **Compilers;** *Runtime environments;* •**Computing methodologies** → *Parallel programming languages;* •**Theory of computation** → *Program analysis;*

1 INTRODUCTION

Task-based parallelism is one of the most fundamental parallel abstractions in common use today [1], with applications in areas ranging from embedded systems, over user-facing productivity and entertainment software, to high performance computing clusters. It provides a convenient programming model for developers, and is available in the majority of mainstream programming languages, parallel extensions, and libraries.

While relatively easy to implement and use, achieving good efficiency and scalability with task parallelism can be challenging. Consequently, it is the subject of ongoing research, and several large projects seek to improve the quality of its implementations. Of particular interest are the efficient scheduling of tasks in ways which optimally use the underlying hardware architecture [2, 11], and research into reducing runtime overheads by e.g. carefully avoiding creating more tasks than necessary [9]. What is common to most research in this area is that it is performed at a library and runtime system level and focuses primarily or exclusively on the *dynamic* behavior of a program. For example, a runtime system might monitor the execution of an algorithm and continuously adjust its scheduling policy based on an active feedback loop [3].

Although these types of approaches have proven very successful and seem inherently suitable for task-parallel programs which might have highly input-data-dependent control flow, they come with some drawbacks: i) they can fundamentally not manipulate settings which need to be fixed at compile time, e.g. because they modify the layout of data structures in memory; ii) dynamic monitoring at the library level can never fully exclude any possible future program behavior, preventing some types of optimizations; and iii) any type of feedback loop will induce some degree of runtime overhead. While its effect can be minimized by careful implementation, even just performing some additional jumps and branching to check whether any adjustments should be performed has a measurable impact in very fine-grained scenarios.

In order to mitigate these drawbacks, we propose a set of static analyses designed to determine features of a task-parallel program that can be used to directly adjust the execution parameters of a runtime system. This approach is orthogonal to runtime optimizations, and can be combined with them in order to find an initial configuration – parts of which might be further refined during program execution. Our concrete contributions are as follows:

- An overall method determining task contexts within a parallel program, performing analyses on each of them, and aggregating their results in order to derive a set of compile-time parameters for a parallel runtime system.
- A set of novel task-specific analyses to determine code features which significantly influence parameter selection, such as the parallel structure or granularity of execution.
- An implementation of this approach within the Insieme compiler and runtime system [8], targeting a set of four runtime parameters.
- An evaluation of our prototype implementation on 12 task-parallel programs on a shared-memory parallel system with up to 64 hardware threads.

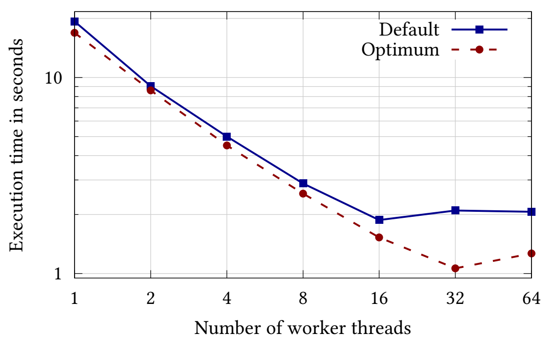

Figure 1: Execution time of the *Strassen* benchmark.

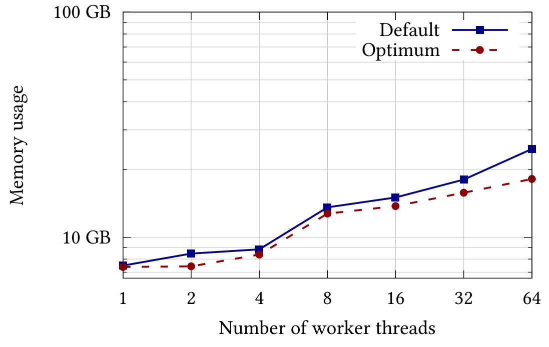

Figure 2: Memory consumption of the *Strassen* benchmark.

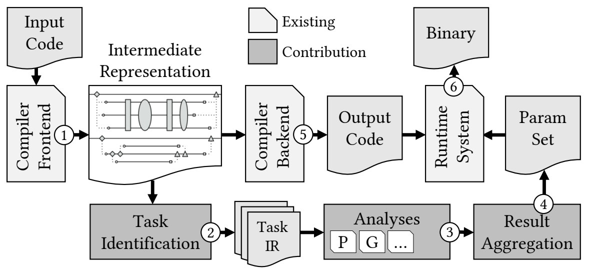

Figure 3: Overview of our method.

The remainder of this paper is structured as follows. We first provide some measurements illustrating the potential improvements possible by optimal parameter selection in order to motivate our approach in Section 2. Section 3 describes our method, including the overall approach, the targeted runtime parameters, and each compiler analysis. The results of our prototype implementation are discussed in Section 4. An overview of related work is provided in Section 5 before concluding the paper.

2 MOTIVATION

Prior to investing the effort required to implement our envisioned method, we estimated the potential gain which might be realized by such a system. To accomplish this goal, we fully explored the runtime parameter space outlined in Section 3.1.2 by exhaustive benchmarking. The hardware and software setup as well as the experimental procedure were the same as for our final evaluation runs, and details concerning these are provided in Section 4.1.

Figure 1 depicts a comparison between the default compile-time parameter configuration for the *Strassen* matrix multiplication benchmark, and the optimum determined by exhaustive search. Note that the chart is in log-log scale, and that with 32 threads the optimal configuration is almost twice as fast as the default. Clearly, the advantage increases with larger degrees of parallelism – a behavior that will be confirmed across all benchmarks in our later experiments, and which is a manifestation of the intuitive idea that the parallel runtime system becomes a progressively larger factor in performance with higher thread counts.

Since one of the runtime parameters we identified as candidate for static tuning primarily influences memory consumption, Figure 2 depicts a similar comparison for this aspect of performance. The relative advantage is lower, but still significant, reaching 36% at 64 threads.

Across the benchmarks described in Section 4.2, *Strassen* is an average example in terms of optimization potential with optimal static parameter selection. As such, a maximum improvement by a factor of 1.97 and 1.36, for execution time and memory consumption respectively, is a very encouraging sign for our approach.

3 METHOD

An overview of our proposed method is provided in Figure 3. Initially, a given task-parallel C or C++ program is translated to a parallelism-aware compiler intermediate representation by the existing compiler frontend ①. Subsequently, as a first pass in our approach, the full lexical extent of each group of tasks is determined, and the code fragments identified are stored for future analysis ②. Several specialized analyses are then performed for each such code fragment ③. The results of these are aggregated, and used to determine parameter settings for the parallel runtime system ④. The compiler backend generates some output code for the task parallel program ⑤, which, together with the automatically configured runtime system, builds the final output binary ⑥.

3.1 Runtime System

In this section, we provide an overview of the runtime system our prototype implementation is based on, as well as the set of parameters explored in this work. While these parameters are specific to our runtime system, similar parameters and concerns exist for all task-parallel systems we are aware of. Crucially, *our general approach of task-specific static analysis for determining per-program compile-time parameter settings is equally applicable to other runtime systems*, and could also be extended to cover a larger set of parameters than the one implemented in this proof-of-concept.

3.1.1 Runtime System Background. The Insieme runtime system which this work is based on is designed to enable low-overhead task-parallel processing. At a basic level, its implementation includes a set of *workers* – generally one per hardware thread – maintaining a local deque of *work items*, which are distributed in a work-stealing manner. These work items correspond to tasks in languages such as Cilk, but provide additional features, including the ability to

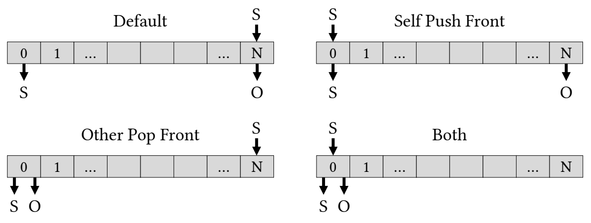

Figure 4: Behavior of available queue policies.

allow for work ranges with runtime-directed splitting, binary multi-versioning [13], and annotation of meta-data by the compiler [14].

This runtime system has been previously demonstrated [13] to outperform many widely-used implementations of recursive task parallelism, and match or exceed the performance of more optimized and specialized frameworks including Cilk+.

3.1.2 Runtime Parameters. We will now describe the set of parameters explored in this work, including their effect on the behavior of the runtime system.

Queue Policy. The queue policy governs how the per-worker deques are used by the runtime when new tasks are generated or a worker is looking for a task to execute next. By default, newly generated tasks are inserted at the end of the executing worker's deque, while a worker initially looks at the front of its deque in order to find new tasks to execute. If its own deque is empty, it will try to steal a task from the back of another worker's deque.

The position where newly created tasks are inserted and from where tasks are stolen from other worker's queues can be configured, and thus our runtime can operate with a total of four different queue policies, as shown in Figure 4. In the illustration, S refers to the worker itself while O refers to some other worker operating on a remote deque during a stealing operation.

The queue policy is expected to impact performance in three major ways:

- Whether newer or older tasks are stolen will significantly influence the granularity of the task – and how many further sub-tasks it might spawn – for recursively parallel algorithms which follow a divide-and-conquer pattern.
- If a calculation is data-intensive, workers executing the most recent task they generated can lead to improvements in cache re-use, especially if e.g. parent tasks make use of the data their children processed.
- When tasks are very fine grained and produced frequently, a large number of accesses being focused on one end of the deque can lead to lock congestion.

Queue Size. The size of the per-worker deques determines the maximum number of work items which can be held at any point, per worker. In this context, it is important to note that the Insieme runtime system performs lazy task generation [9], as is common for high-performance implementations of task parallelism. That is, if a worker's deque is full, a newly launched task will be immediately executed sequentially, rather than generating the full set of work item data and registration information required for its eventual asynchronous execution and synchronization.

Due to this behavior – which is essential in order to achieve high performance with fine-grained tasking – selecting an effective queue size for a given problem requires a trade-off between two conflicting goals. On the one hand, the chosen size needs to be sufficiently large in order to avoid a situation in which there are few or no remaining tasks available in the system, leading to a starvation of workers and inefficient parallel execution. On the other hand, choosing a shorter queue can reduce the overhead incurred for work item generation while a sufficient number of them is available and/or more are being generated at a good pace.

We investigated queue sizes of 4, 8, 16, 32, 64 and 128, with 16 being the default in the Insieme runtime system.

Event Table Buckets. For use cases which unavoidably require some type of global knowledge or bookkeeping, such as work item synchronization, the Insieme runtime system implements a thread-safe *event table* based on open hashing and fine-grained locking. Since any delay in synchronization will lead to low worker utilization, the efficient implementation of this table is of utmost importance, particularly for high degrees of shared-memory parallelism. The default event table bucket count in the Insieme runtime system is 97. We also conducted experiments with the larger prime numbers 1021, 64567 and 256019.

The number of buckets in the event hash table should be chosen based on the amount of active tasks which are expected to require synchronization at the same time. If there are few such tasks, a small bucket count will allow for more effective cache utilization. However, if the number of active tasks at any point becomes significantly higher than the number of buckets, the open hashing implementation will become significantly less effective, as the expected event registration and triggering performance drops from $O(1)$ to $O(N)$.

Stack Size. Starting the execution of a new work item requires allocating a stack frame for this task. While a task-parallel runtime system can potentially grow the stack based on demand, in a large-scale user-level threading scenario this quickly becomes a significant performance hurdle and source of complexity. Therefore, a simple solution in use in several existing systems, including the Insieme runtime, is initially allocating a large stack (i.e. equal to the OS maximum). By analyzing the per-task stack requirements, the initial stack size can be reduced for programs only storing a small amount of data on the stack, decreasing memory requirements – and potentially increasing performance e.g. in case the new size is small enough to fit into per-thread storage provided by the memory allocator in use.

In our evaluation, we executed the programs with different stack sizes in powers of 2, ranging from 16 kB to 8 MB – the latter representing the conservative default setting in the Insieme runtime system.

3.2 Compiler Analysis

A central component of our approach are a set of compiler analyses explicitly designed to determine information about task-parallel codes which is relevant for configuring runtime system parameters. In this section, we will first provide a short overview of the compiler infrastructure we chose to implement these analyses, and then describe each of them in detail.

Table 1: INSPIRE constructs for task parallelism

Construct / Type	Semantics
parallel (job) → thread_group	Launches a new parallel job with the supplied *job* description, returning a thread group to synchronize on it.
job (range, f) → job	Creates a new job with the given *range*, executing the lambda f of type () → *unit*.
merge (thread_group) → *unit*	Synchronizes the execution of the given thread group, waiting for it to finish before continuing the current thread.
merge_all () → *unit*	Synchronizes the execution of *all* thread groups launched by the current thread.

3.2.1 Compiler Background. In order to accomplish the analyses required for our approach, a high-level intermediate representation (IR) with native parallelism-awareness is advantageous. We chose the Insieme research compiler infrastructure as its INSPIRE IR [7] is designed to fully capture semantics relevant for parallelism from a variety of input languages.

A full description of this IR is beyond the scope of this paper, and we refer the interested reader to the description by Jordan et al. [7]. For the purpose of our analysis discussion, some features are of particular importance:

- Task-based parallelism is primarily encoded by the set of constructs listed in Table 1, with an informal description of their semantics. Note that the *unit* type is the equivalent of *void* in C-like languages, i.e. representing the absence of a return value.
- Built-in operands, functions in the original input program, and functions generated during front-end processing and optimization are encoded as *Lambdas*, and referred to using *LambdaReferences* in a recursive context.
- Any data stored on the stack is allocated in *Declaration* nodes. This includes variables in declaration statements, as well as function call arguments and return values.
- All operations and analyses on INSPIRE are inherently whole-program and inter-procedural. As task execution generally requires capturing of context data and passing an executable parameter to a higher-order function, local analysis does not provide useful insight for our use case.

In addition to these features, some terminology related to two fundamental concepts will be referred to throughout the remainder of this section:

IR Nodes are the basic components which the IR is comprised of. Each node n may have an arbitrary number of child nodes C^n forming the sequence $[n_1, n_2, ..., n_N]$, and the directed acyclic graph (DAG) of nodes starting from the *main* lambda represents an entire program.

Starting from some node n, we write n_i to refer to the ith child node of n, with further child nodes indicated by additional indices in a tuple.

IR Addresses represent a specific position within a program or smaller IR fragment. They consist of a *root node* and a *path*, with the latter containing a list $[i_1, i_2, ..., i_D]$ of child node indices. For a path length of D, $D - 1$ nodes are traversed starting from the root node before arriving at the node pointed to by the given address. Therefore D determines the *depth* of an address.

When referring to an address, the sequence of nodes indicated by the indices starting from and including the root node r is designated as the *address node sequence* $[r, r_{(i_1)}, r_{(i_1,i_2)}, ..., r_{(i_1,i_2,...,i_D)}]$. In the context of a particular address, r_{i_j} is the *parent* node of $r_{(i_j,i_{j+1})}$.

3.2.2 Common Operations. Before describing individual analyses, we will first define a set of common operations which simplify the formulation of our algorithms.

call_of(f, A) Refers to any call of the *Lambda* or *LambdaReference* f with the list of argument expressions A.

all_calls_of(n, f) This operation returns a set of all addresses rooted at node n to calls of the construct f in any child node of n, at arbitrary depth, regardless of their arguments.

call_of_ref(l) Refers to any call of the lambda l by *LambdaReference*, regardless of its arguments.

def_of(l) Refers to the definition of a lambda with the *LambdaReference* l.

loop(i, b, h) Refers to any type of loop with i iterations, the body b and header h. The loop header includes all the nodes to check the loop boundaries and update the loop counter.

declaration(τ, i) Refers to a declaration node of type τ with the initialization expression i.

reverse_sequence(a) For address a with root r and path $[i_1, i_2, ..., i_D]$, returns the address sequence $[r_{(i_1,i_2,...,i_D)}, ..., r_{(i_1)}, r]$.

all_leaf_addresses(n) Returns the set full of all leaf addresses (with $|C^a| = 0$) reachable from node n.

is_builtin(f) Checks whether the construct f is a built-in construct.

The description of our analyses based on these primitives matches the implemented semantics, but often does not match the implementation exactly. Various optimizations aimed at reducing the execution time of the compiler, such as result caching and early pruning, increase the complexity of describing an algorithm and are therefore omitted in the depictions in this paper.

3.2.3 Task Context Identification. Identifying the lexical IR fragments relevant for each individual task is a prerequisite for all subsequent analyses, and listed as step ② in the overview provided in Figure 3. The input to this step is a full program in INSPIRE, and its outputs are root IR nodes of the task code fragments identified.

Algorithm 1 depicts the task context identification process. Initially, a set T of the addresses of all *parallel* calls with a range of $[1, 1]$ – that is, task invocations – is determined. The node address sequences for these are then traversed bottom-up until the first original program function is found, and the addresses of those are then added to T' which is the returned set. The bottom-up traversal is necessary to include the entire original calling context of the task for future analysis, as it might have been wrapped in additional built-in calls during front-end translation to INSPIRE.

3.2.4 Determining the Parallel Structure. An essential feature of each task context which heavily influences good decision-making, in particular for the Queue Policy parameter, is its *parallel structure*.

Algorithm 1 Task Context Identification

m root node of the *main* lambda

```
1:  $P \leftarrow$ all_calls_of($m$, $parallel$)
2:  $T \leftarrow \{parallel(job(r, f)) \in P \mid r = [1, 1]\}$
3:  $T' \leftarrow \{\}$
4:  for all $t \in T$ do
5:      for all $n \in$ reverse_sequence($t$) do
6:          if $\exists f, A \mid n =$ call_of($f, A$) $\wedge \neg$is_builtin($f$) then
7:              $t \leftarrow f$
8:              break
9:      $T' \leftarrow T' \cup \{t\}$
10: return $T'$
```

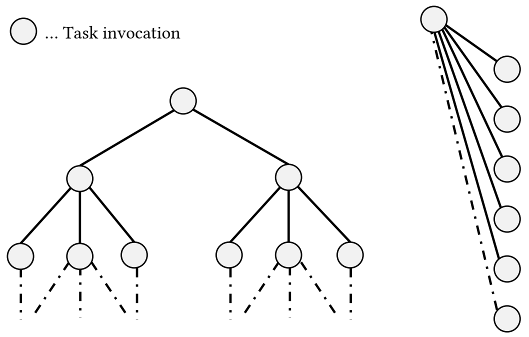

Figure 5: Fundamental parallel program structures.

Figure 5 illustrates two fundamental types of parallel structures that can be encountered in task-parallel programs. A *recursive* structure indicates that individual tasks invoke self-similar subtasks, while a *loop-like* structure is present if task invocation occurs within an outer loop. Note that both can be present at the same time, if a program spawns recursive tasks within a loop in the same or a mutually recursive function. It is also possible in theory for a task-parallel program to be neither recursive nor loop-like in structure; in practice, such a program is unlikely, as its degree of parallelism would be statically determined and independent of its input data.

Algorithm 2 determines the set of recursive parallel paths within a given task invocation context. It traverses the address node sequence of each possible leaf address bottom-up, noting the call site of a lambda invoked by reference. If such a call has occurred, and a *parallel* call exists on the path between it and the definition of the callee, then the path performs a recursive parallel invocation. The algorithm for determining parallel invocations within loops is quite similar, and not listed separately due to space concerns. Instead of searching for definitions of recursively invoked lambdas, it looks for loop constructs along the path from each task invocation to the main entry point of the program.

Figure 6 illustrates a simplified example of an INSPIRE address tree for a task-parallel program. The definition of lambda *foo* at Ⓐ will be identified as the task context by Algorithm 1, as it is the innermost non-built-in lambda containing a parallel invocation

Algorithm 2 Determine Recursive Parallel Paths

t root node of the task context

```
1:  $P \leftarrow \{\}$
2:  for all $a \in$ all_leaf_addresses($t$) do
3:      $l' \leftarrow \perp$
4:      $p \leftarrow \perp$
5:      $c \leftarrow \perp$
6:      for all $n \in$ reverse_sequence($a$) do
7:          if $\exists l \mid n =$ call_of_ref($l$) then
8:              $l' \leftarrow l$
9:              $c \leftarrow n$
10:         else if $c \wedge \exists A \mid n =$ call_of($parallel, A$) then
11:             $p \leftarrow \top$
12:         else if $c \wedge p \wedge n =$ def_of($l'$) then
13:             $P \leftarrow P \cup (n, c)$
14:             break
15: return $P$
```

```
LambdaExpr main
└ ...
  └ LambdaExpr foo Ⓐ
    └ ...
      └ CompoundStmt
        ├ ...
        ├ IfStmt
        │ ├ CompoundStmt
        │ │ ├ ...
        │ │ ├ CallExpr
        │ │ │ └ LambdaReference foo Ⓑ
        │ │ └ ...
        │ └ CompoundStmt
        │   ├ ...
        │   └ CallExpr Ⓒ
        │     ├ parallel
        │     └ JobExpr
        │       ├ JobRange [1..1]
        │       └ LambdaReference foo Ⓓ
        ├ ...
        └ CallExpr
          └ mergeAll
```

Figure 6: Example INSPIRE address tree structure.

with a job range of $[1, 1]$ Ⓒ. Algorithm 2 will evaluate all paths from each leaf. The path starting at Ⓑ demonstrates the necessity for checking for a parallel invocation on the closed recursion cycle: it is recursive and within the parallel context, but not an instance of parallel recursion. Conversely, the path starting at Ⓓ contains a call to *parallel* at Ⓒ, and will be correctly detected by the algorithm.

3.2.5 Task Granularity Estimation. Knowledge of the expected granularity of tasks – that is, the average time the program spends between interactions with the runtime system, such as task creation and synchronization – is a highly significant feature for scheduling decisions. While a completely accurate static analysis of this granularity is generally infeasible due to e.g. unknown input problem

Algorithm 3 Effort Estimation

B	effort mapping function for built-ins

```
function EFFORT($n$)
    $e \leftarrow 0$
    if $\exists f, A \mid n = \text{call\_of}(f, A)$ then
        for all $\alpha \in A$ do
            $e \leftarrow e + \text{EFFORT}(\alpha)$
        if n = call_of_ref($f$) then
            return $e$
        if is_builtin($f$) then
            return $e + B(f)$
        return $e + \text{EFFORT}(f)$
    if $\exists i, b, h \mid n = \text{loop}(i, b, h)$ then
        return $i * (\text{EFFORT}(b) + \text{EFFORT}(h))$
    for all $c \in C^n$ do
        $e \leftarrow e + \text{EFFORT}(c)$
    return $e$
```

sizes, even having a rough indication at compile time of whether tasks will be particularly fine- or coarse-grained is helpful.

Algorithm 3 performs a static *effort estimation* on an arbitrary INSPIRE node *n*. By default, it simply traverses all child nodes (line 13). Function calls and loops are handled specifically. For all function calls, initially the effort for evaluating their arguments is determined. Built-ins – such as arithmetic operations, array subscripts or assignments – are mapped to predefined values supplied in an effort mapping function *B*. Other calls are evaluated by recursive invocation of the algorithm. For loops, the effort determined for each iteration is multiplied by the number of iterations. In case the iteration count cannot be determined statically, we currently assume a fixed estimate of 100 iterations. While this branch-invariant approach which ignores dynamic loop iteration counts will be highly inaccurate when trying to make e.g. absolute execution time predictions, in our use case some indication of granularity proves sufficient to improve compile-time decision making. Including better analysis for loops with dynamic iteration counts could be part of future work.

3.2.6 Stack Size Estimation. The final analysis for our parameter selection provides an estimation of the required stack frame size of a given task context. As explained in Section 3.1.2, a good stack size choice can improve both performance and particularly memory consumption for programs generating many small tasks.

As Algorithm 4 illustrates, stack size estimation for a given task context can be expressed quite succinctly due to the properties of INSPIRE. All stack memory allocations derive from *declaration* nodes, which are handled in the initial branch of the STACK_SIZE function. This function requires a map *S* from types to their size in bytes, and a constant recursion estimate ϕ as its inputs, and builds up a set of visited references during its execution. It returns a pair of two values: the stack requirement at node *n* itself and the total stack requirement for the full sub-tree rooted at that node. The basic idea is that, for all nodes, the local stack requirements are the sum of the local stack requirements of all child nodes, while the total stack requirement is the maximum of all its child stack requirements. Thanks to the IR structure, this simple principle accurately covers various cases such as function call arguments, compound statements, and control flow.

Algorithm 4 Stack Size Estimation

t		root lambda of the task context
ϕ		constant recursion estimate factor
S		type size mapping
V	{}	set of visited references

```
function STACK_SIZE($n$)
    if $\exists \tau, i \mid n = \text{declaration}(\tau, i)$ then
        $s \leftarrow S(\tau)$
        return $(s, s + \text{STACK\_SIZE}(i))$
    $(p, q) \leftarrow (0, 0)$
    for all $c \in C^n$ do
        $(p', q') \leftarrow \text{STACK\_SIZE}(c)$
        $p \leftarrow p + p'$
        $q \leftarrow max(q, q')$
    if $\exists l \mid n = \text{call\_of\_ref}(l)$ then
        if $\text{def\_of}(l) \neq t \wedge l \notin V$ then
            $V \leftarrow V \cup l$
            $(p', q') \leftarrow \text{STACK\_SIZE}(\text{def\_of}(l))$
            return $(p + p' * \phi, q + q' * \phi)$
    return $(p, q)$
```

3.3 Result Aggregation

As all parameters we currently study must be set once for the entire runtime system – rather than per-task – the results derived by our per-task analyses need to be aggregated before they can be used to derive parameter settings. The correct way to perform this aggregation depends on the analysis in question and its use case.

3.3.1 Parallel Structure. The aggregate number of recursive parallel paths is chosen as the *minimum* across all task contexts in the program. Since this number indicates whether or not tasks produce additional work, which impacts parameters such as queue size and policy, assuming that all tasks produce further tasks when this is not necessarily the case can cause severe starvation issues. The opposite – under-estimating the amount of tasks generated – can cause additional overhead, but not a sudden and severe performance drop-off. The same reasoning applies to loops, and the whole program is only treated as featuring loop-like parallelism if all of its task contexts do.

3.3.2 Granularity. For granularity estimation across the whole program, simply choosing the *mean* granularity across all task contexts is intuitive and works well in practice.

3.3.3 Stack Size. As all work items instantiated during the program's execution need to be accommodated, the *maximum* of all individual estimates is chosen. It is also rounded up to the next power of two for alignment purposes, and a minimum of 16 kB is applied.

3.3.4 Deriving Parameter Values. While the one-to-one mapping from the stack size analysis result to the actual runtime parameter is obvious, defining the queue policy, queue size, and number of event table buckets based on our analysis results requires some strategy. For our prototype, this mapping was derived as a simple decision tree per parameter, based on empirical experience. Note that in the following description, ρ represents the number of recursive parallel invocations detected, λ lists the number of loop-like parallel invocations, and e refers to the per-task granularity or *Effort* estimated by our analysis. Actual values for these analysis results are presented in Table 3 in the evaluation section.

$$\text{queue\_policy}(\rho, \lambda, e) = \begin{cases} PF & \text{if } \rho > 5 \vee (1\text{M} < e \leq 1\text{T}) \\ DEF & \text{otherwise} \end{cases}$$

$$\text{queue\_size}(\rho, \lambda, e) = \begin{cases} 128 & \text{if } \rho = 0 \wedge \lambda > 0 \\ 8 & \text{else if } 1\text{M} < e \leq 1\text{T} \\ 4 & \text{otherwise} \end{cases}$$

$$\text{table\_buckets}(\rho, \lambda, e) = \begin{cases} 256019 & \text{if } \rho > 0 \\ 97 & \text{otherwise} \end{cases}$$

For the queue policy parameter, the "Self Push Front" (PF) strategy is chosen over the default if a benchmark features many recursive tasks or is of medium granularity. The remaining two queue policies mostly mirrored the results we obtained for the two used by our selection strategy. A large queue length of 128 is advantageous for loop-like parallel programs, while very fine-grained recursive ones favor a very short queue as new tasks are generated rapidly. Finally, the optimal number of event table buckets depends purely on whether recursive tasks are present – if so, a far larger number of synchronization operations might be pending.

4 EVALUATION

4.1 Evaluation Platform and Setup

Our evaluation platform is a quad-socket system equipped with four Intel Xeon E5-4650 processors, each offering 8 cores (16 hardware threads) clocked at a frequency of 2.7 GHz. The software stack on this system is based on CentOS 6.7 running kernel version 2.6.32-573. All our binaries were compiled with GCC 5.1.0 using -O3 optimizations to approximate a realistic production scenario.

For parallel execution, the thread affinity in all benchmark runs was fixed using a fill-socket-first policy, in order to improve the reliability of measurements and minimize variance. All reported numbers and figures are based on medians over seven runs. Memory consumption is measured as the maximum resident set size across the entire execution of a given benchmark.

4.2 Benchmarks

Table 2 lists the benchmarks we used to validate and evaluate our approach, along with their origin as well as their structure, granularity and parameters. Most benchmark code versions are taken directly from the Barcelona OpenMP tasks suite [4], while the QAP2 benchmark was introduced in the Inncabs [12] suite. Both of these publications describe each involved benchmark in some detail. The structure (loop-like, recursive balanced or recursive unbalanced) and granularity indicators in Table 2 are sourced from these publications, and based on human judgment and measurements of each code.

Table 2: Benchmark Overview

Benchmark	Origin	Struct.	Granularity	Parameters
Alignment	AKM	loop	coarse	prot.100.aa
Delannoy	-	rec. b.	very fine	11
FFT	Cilk	rec. b.	variable	-n 16777216
Fib	-	rec. b.	very fine	-n 35
Floorplan	AKM	rec. u.	fine	input.20
Health	BOTS	loop	moderate	medium.input
NQueens	Cilk	rec. u.	moderate	-n 14
QAP2	Inncabs	rec. u.	fine	chr15a.dat
Sort	Cilk	rec. b.	variable	-n 134217728
SparseLU	BOTS	loop	coarse	-n 50 -m 100
Strassen	Cilk	rec. b.	moderate	-n 4096
UTS	UNC	rec. u.	variable	test.input

Table 3: Benchmark Properties (Analysis)

Benchmark	ρ	λ	Effort	Stack
Alignment	0	1	2.6 T	8 M
Delannoy	3	0	131.0	16 k
FFT	27	1	970.0 M	256 k
Fib	2	0	52.0	16 k
Floorplan	1	1	39.0 G	2 M
Health	1	1	33.0 G	32 k
NQueens	1	1	601.0 M	2 M
QAP2	1	1	13.0 M	16 k
Sort	6	0	2.4 G	32 k
SparseLU	0	3	3.3 P	16 k
Strassen	7	0	282.0 G	256 k
UTS	1	1	12.0 T	2 M

4.3 Quality of Analysis

Before presenting execution time and memory usage improvements achieved by our prototype implementation, we will first evaluate the accuracy of our analyses on the given set of benchmarks. Table 3 lists the parallel structure, effort estimation, and stack size properties determined by our analyses.

Comparing ρ and λ with the manual structure categorization provided in Table 2 reveals interesting correlations:

- The only benchmarks with $\rho = 0$ are categorized as loop-like, confirming this result.
- While *Health* is categorized as "loop-like", inspection of the source code confirms the analysis result: there is an indirect recursive invocation within the loop. Here, our analysis provides a more exact result than a cursory manual inspection.
- Recursive benchmarks with $\rho > 1$ or $\lambda = 0$ are likely to have a balanced task workload, while the ones with $\rho = 1$ are likely unbalanced.

The final observation is of particular interest, as the balance or imbalance of recursive workloads is not something we expected to be indicated by static analysis. Clearly, load imbalance on an individual task level often occurs due to input data dependence, which appears to commonly manifest in a variable number of loop iterations containing task invocations.

The *Effort* column in Table 3 lists the results of our granularity analysis (Algorithm 3). Comparing this to the manual categorization, we observe the following:

- The benchmarks assumed to be of "very fine" granularity are also the most fine-grained according to analysis, by several orders of magnitude.
- Benchmarks categorized as coarse-grained are in the peta- and tera-scale range and at the upper end of values according to analysis.
- *Floorplan* and *UTS* feature relatively high granularity values compared to their manual classification based on measurements. Inspecting their source code reveals that this is due to their recursive invocations containing loops with input-dependent iteration counts which are very low with the problem sizes used in our evaluation.

Overall, while not as exact as the categorization of parallel structure, our granularity analysis still provides a guideline which correlates well with the actual program behavior in most cases. Fully accurate granularity prediction at compile time remains impossible for realistic programs with dynamic input data.

Finally, the *Stack* column in Table 3 lists the results of our stack size estimation, in bytes. The most important quality metric for these results is the ability for each benchmark to complete without running out of stack space, which is accomplished for all results. *Alignment* is estimated to require a full 8 MB of stack size per task – an investigation of its source code reveals that this is explained by it allocating multiple large arrays on the stack in recursive calls.

4.4 Benchmark Performance Evaluation

While our evaluation so far has shown that our analyses provide good approximations of important task features, we have not yet demonstrated that these features are actually useful for their intended purpose of optimizing runtime settings. In this section, we apply our full method to the benchmarks presented in Section 4.2 and measure the resulting performance.

4.4.1 Execution Time. Figure 7 depicts the execution time using the optimized parameter settings determined by our approach ($T_{\text{optimized}}$) relative to the execution time using default settings (T_{default}). Note that the default settings in this comparison are the out-of-the-box defaults of the Insieme runtime system, which are highly competitive with several widely-used task-parallel systems [13]. Results from all benchmarks are summarized in a box plot, which allows us to illustrate the overall effectiveness of our approach without missing important outliers, particularly if they were to occur in the negative direction. These results allow for the following observations:

- The lower quartile is always above 1.0, indicating that our approach performs as well or better than the default for at least 75% of our benchmarks, at all degrees of parallelism.

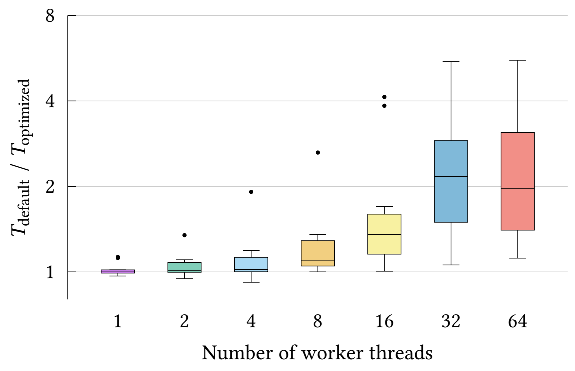

Figure 7: Overall execution time comparison.

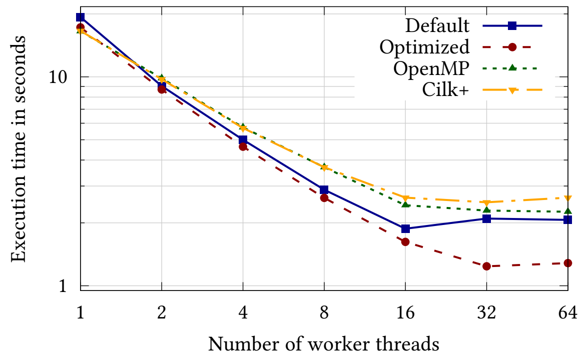

Figure 8: Performance charts for the *Strassen* benchmark.

- Starting from 8 worker threads and at all higher degrees of parallelism, all benchmarks obtain at least some improvement in performance. The geometric mean factor across all benchmarks and thread counts is 1.39.
- The largest performance increase is obtained at 32 worker threads, where our optimized versions perform more than twice as fast as the defaults for most benchmarks.
- Overall, the lowest value encountered is 0.91, indicating a 9% performance loss. This occurs for the QAP2 benchmark with four hardware threads.

The trend of increasing performance gains with higher worker thread counts can be attributed to two reasons. For one, with higher degrees of parallelism the effectiveness of the runtime system in facilitating task creation, scheduling and synchronization gains more prominence as a factor in overall program performance, and these operations can be optimized by good parameter choices. For another, the default runtime parameter settings also appear to be more tuned for smaller shared-memory systems.

In order to illustrate that the comparative basis chosen for this performance evaluation is meaningful, we revisit our motivational *Strassen* example in Figure 8, while also adding measurements for the standard GCC OpenMP implementation as well as Cilk+. As shown, performance using the default parameter settings is competitive with – and in fact, at 4 or more threads, superior to

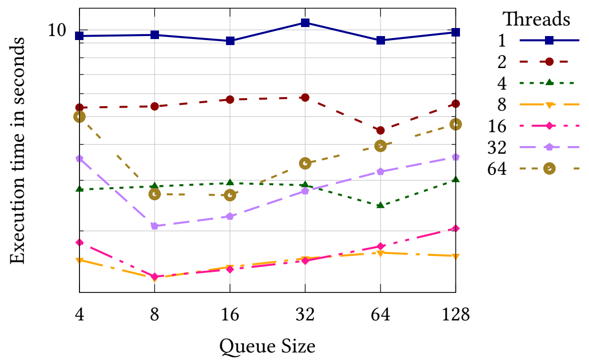

Figure 9: QAP2 execution times with varying queue size.

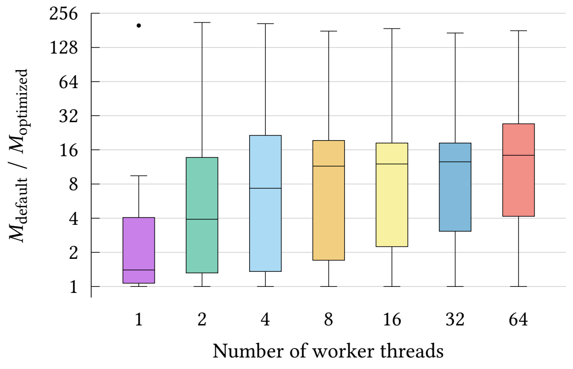

Figure 10: Overall memory consumption comparison.

– these industry-standard solutions, and our statically optimized parameter set further improves on this result, coming close to parity with the exhaustively determined optimum in Figure 1.

Regarding the small performance losses incurred for a few benchmarks with two and four worker threads, investigating the causes for these in more detail reveals that the affected benchmarks are those which benefit greatly from data cache locality across parent and child tasks. For larger thread counts and particularly once more than a single socket is used, other concerns dominate performance. Figure 9 illustrates how this difference in optimal parameter selection between single- and multi-socket execution manifests in diverging patterns in practice. Currently, we do not perform any analysis which tries to determine the impact of stack memory access locality for a benchmark. There is an opportunity for future work in this area to eliminate the cases of performance degradation, however, as it is relatively minor and limited to a small number of specific benchmarks and thread counts, the significant complexity of such analysis might not be justifiable.

4.4.2 Memory Consumption. Since one of the parameters we optimize primarily affects memory consumption, we also evaluated this aspect of runtime system performance. Figure 10 provides this overview, using the same methodology as employed for Figure 7. We observe the following:

- For all thread counts, no benchmarks suffer from an increase in memory consumption. However, a few benchmarks also show no improvement at all.
- There is an increase in the impact of our optimizations with increasing thread counts, but the correlation is not as high as it is for execution times.
- The maximum improvement is very high, at a factor of more than 100.

All of these observations can be explained by considering a few factors. First of all, some programs feature heavy heap memory use for their own data, or require a large stack size, which explains why no improvement can be achieved for some benchmarks regardless of the level of parallelism.

The fact that improvements scale with the degree of parallelism initially but flatten out soon is due to the behavior of lazy task generation: initially, more parallelism will lead to significantly more tasks being generated, and thus more stacks allocated, but this effects becomes less pronounced after a certain point. Finally, the reason for the extremely high factors achieved in some benchmarks is due to the default behavior of the runtime system: without any static knowledge, it provides each task with an initial stack frame of 8 MB to ensure correct execution. For benchmarks with extremely small stack and heap data sizes such as *Fib*, reducing that per-task allocation down to e.g. 16 kB will massively decrease overall *relative* memory consumption. Systems such as Cilk which implement a cactus stack layout [6] would not benefit as dramatically from this optimization.

5 RELATED WORK

There is a very large body of work dealing with the optimization of task-parallel programs at runtime, often at the library level. A small subset of these works was referred to in Section 1. As noted there, these types of optimizations are orthogonal to and can be combined with our method. In this section, we will focus on research which performs runtime parameter tuning with a parallelism-specific compiler analysis component.

Tick and Zhong [16] propose a combined compile-time and runtime method to improve performance and reduce execution overheads caused by too small-grained parallel tasks. A compiler analysis produces estimator functions for parallel tasks, which can then be evaluated at execution time to improve task scheduling. This matches a single component of our analysis approach, which estimates granularity, however we also provide analyses for the parallel structure and memory footprint of individual tasks, and take these into account at compile time rather than during execution. In a similar work [13], we leveraged a compiler component to control task granularity, but rather than providing estimates, granularity was actively adjusted by multiversioning of task functions.

Vuduc et al. [17] forward compiler analysis results to the runtime in the form of a decision function, in order to select among several versions of the same algorithm depending on input features. However, their optimization affects program- and algorithm-specific decision making during execution time, while we focus on general runtime system decisions made at compile time.

In the context of software distributed shared memory systems, Dwarkadas et al. [5] implement a combined compile-time and runtime method. The compiler component analyses programs to reason about data access patterns and forwards this information to the runtime part of the system. This additional information enables the runtime system to aggregate communication and synchronization operations, and thus reduce runtime overheads. Another approach combining a custom compiler component with a runtime library is described by Nikolopoulos et al. [10]. Their compiler analyzes OpenMP programs and evaluates the thread memory reference semantics. The gathered information enables the runtime system to accurately perform page migrations to improve program throughput independently of the operating system's memory page placement strategy. Both of these papers focus on data access patterns and data parallelism, which is not currently part of our analyses but could be treated in our general framework.

One of our previous works [14] leverages static analysis of programs for improved runtime behavior in relation to program characteristics. However, it focuses entirely on loop parallelism and one specific optimization. Conversely, all analysis and optimization in this work applies primarily to task-parallel programs. Recently, we investigated semantics-aware compilation of the C++11 standard library for task-based parallelism [15]. While an ad-hoc task classification scheme was employed, this work lacks sophisticated compiler analysis, features a very limited set of parameters, and only supports a single task type per program.

6 CONCLUSION

We have presented a method for optimizing parameters of task-parallel runtime systems by performing a set of compiler analyses – specifically designed to classify and characterize tasks – on their input programs. As our approach is *entirely static*, it improves upon common purely dynamic task optimization by being able to manipulate parameters which need to be set at compile time, as well as having the ability to leverage information which is expensive or infeasible to obtain during program execution.

Evaluation of our prototype implementation on a set of 12 benchmarks representing a variety of parallel algorithm structures and granularities demonstrates increasingly significant performance improvements with an increasing degree of parallelism. At 32 threads, a geometric mean improvement in execution time across all benchmarks by more than a factor of 2 is achieved. At the same time, peak memory usage is reduced by over an order of magnitude for fine-grained benchmarks with only very small stack requirements which can be determined statically.

The general method presented here can be extended in several areas which present opportunities for future research. More task context analyses, such as data reuse across parent and child tasks, can be integrated in order to make even more accurate parameter selections. Additionally, the set of runtime parameters being optimized might be extended to increase the potential performance gains. Finally, our current prototype mapping from analysis results to parameter settings can be replaced by a more sophisticated and automated approach.

ACKNOWLEDGEMENT

This project has received funding from the European Union's Horizon 2020 research and innovation programme as part of the FETHPC AllScale project under grant agreement No 671603.

REFERENCES

[1] Krste Asanovic, Ras Bodik, Bryan C. Catanzaro, Joseph J. Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William L. Plishker, John Shalf, Samuel W. Williams, and Others. 2006. *The landscape of parallel computing research: A view from Berkeley.* Technical Report. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley.

[2] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2009. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. (Aug. 2009).

[3] Shimin Chen, Phillip B. Gibbons, Michael Kozuch, Vasileios Liaskovitis, Anastassia Ailamaki, Guy E. Blelloch, Babak Falsafi, Limor Fix, Nikos Hardavellas, Todd C. Mowry, and Chris Wilkerson. 2007. Scheduling Threads for Constructive Cache Sharing on CMPs. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '07).* ACM, New York, NY, USA, 105–115. DOI: http://dx.doi.org/10.1145/1248377.1248396

[4] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade. 2009. Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP. In *2009 International Conference on Parallel Processing.* 124–131. DOI: http://dx.doi.org/10.1109/ICPP.2009.64

[5] Sandhya Dwarkadas, Alan L. Cox, and Willy Zwaenepoel. 1996. An Integrated Compile-time/Run-time Software Distributed Shared Memory System. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII).* ACM, New York, NY, USA, 186–197. DOI: http://dx.doi.org/10.1145/237090.237181

[6] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The Implementation of the Cilk-5 Multithreaded Language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI '98).* ACM, New York, NY, USA, 212–223. DOI: http://dx.doi.org/10.1145/277650.277725

[7] Herbert Jordan, Simone Pellegrini, Peter Thoman, Klaus Kofler, and Thomas Fahringer. 2013. INSPIRE: The Insieme Parallel Intermediate Representation. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques (PACT '13).* IEEE Press, Piscataway, NJ, USA, 7–18.

[8] H. Jordan, P. Thoman, J. J. Durillo, S. Pellegrini, P. Gschwandtner, T. Fahringer, and H. Moritsch. 2012. A Multi-Objective Auto-Tuning Framework for Parallel Codes. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for.* 1–12. DOI: http://dx.doi.org/10.1109/SC.2012.7

[9] E. Mohr, D. A. Kranz, and R. H. Halstead. 1991. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems* 2, 3 (Jul 1991), 264–280. DOI: http://dx.doi.org/10.1109/71.86103

[10] Dimitrios S. Nikolopoulos, Theodore S. Papatheodorou, Constantine D. Polychronopoulos, Jesús Labarta, and Eduard Ayguadé. 2000. UPMLIB: A Runtime System for Tuning the Memory Performance of OpenMP Programs on Scalable Shared-Memory Multiprocessors. In *Languages, Compilers, and Run-Time Systems for Scalable Computers: 5th International Workshop, LCR 2000 Rochester, NY, USA, May 25–27, 2000 Selected Papers*, Sandhya Dwarkadas (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 85–99. DOI: http://dx.doi.org/10.1007/3-540-40889-4_7

[11] Stephen L Olivier, Allan K Porterfield, Kyle B Wheeler, Michael Spiegel, and Jan F Prins. 2012. OpenMP task scheduling strategies for multicore NUMA systems. *The International Journal of High Performance Computing Applications* 26, 2 (2012), 110–124. DOI: http://dx.doi.org/10.1177/1094342011434065

[12] P. Thoman, P. Gschwandtner, and T. Fahringer. 2015. On the Quality of Implementation of the C++11 Thread Support Library. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing.* 94–98. DOI: http://dx.doi.org/10.1109/PDP.2015.33

[13] Peter Thoman, Herbert Jordan, and Thomas Fahringer. 2013. Adaptive Granularity Control in Task Parallel Programs Using Multiversioning. In *Euro-Par 2013 Parallel Processing: 19th International Conference, Aachen, Germany, August 26-30, 2013. Proceedings*, Felix Wolf, Bernd Mohr, and Dieter an Mey (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 164–177. DOI: http://dx.doi.org/10.1007/978-3-642-40047-6_19

[14] Peter Thoman, Herbert Jordan, Simone Pellegrini, and Thomas Fahringer. 2012. *Automatic OpenMP Loop Scheduling: A Combined Compiler and Runtime Approach.* Springer Berlin Heidelberg, Berlin, Heidelberg, 88–101. DOI: http://dx.doi.org/10.1007/978-3-642-30961-8_7

[15] Peter Thoman, Stefan Moosbrugger, and Thomas Fahringer. 2015. *Optimizing Task Parallelism with Library-Semantics-Aware Compilation.* Springer Berlin Heidelberg, Berlin, Heidelberg, 237–249. DOI: http://dx.doi.org/10.1007/978-3-662-48096-0_19

[16] E. Tick and X. Zhong. 1993. A compile-time granularity analysis algorithm and its performance evaluation. *New Generation Computing* 11, 3 (1993), 271. DOI: http://dx.doi.org/10.1007/BF03037179

[17] Richard Vuduc, James W. Demmel, and Jeff A. Bilmes. 2004. Statistical Models for Empirical Search-Based Performance Tuning. *The International Journal of High Performance Computing Applications* 18, 1 (2004), 65–94. DOI: http://dx.doi.org/10.1177/1094342004041293