Journal of Signal Processing Systems (2019) 91:303–320
https://doi.org/10.1007/s11265-018-1356-9

Static Compiler Analyses for Application-specific Optimization of Task-Parallel Runtime Systems

Peter Thoman[1] · Peter Zangerl[1] · Thomas Fahringer[1]

Received: 10 August 2017 / Revised: 3 December 2017 / Accepted: 19 March 2018 / Published online: 24 April 2018

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 overhead and cannot affect parameters which need to be set 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, the memory size of the closure required for task parameters, 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%. To illustrate the impact of our optimizations, we also provide a comparison to current state-of-the art task-parallel runtime systems, including OpenMP, Cilk, HPX, and Intel TBB.

Keywords Compiler · Runtime · Task parallelism · Static analysis · Task granularity · Parallel pattern

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, 18], and research into reducing runtime overheads by e.g. carefully avoiding creating more tasks than necessary [15]. 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 [6].

Although these dynamic approaches have proven very successful and seem inherently suitable for task-parallel

✉ Peter Thoman
petert@dps.uibk.ac.at

Peter Zangerl
peterz@dps.uibk.ac.at

Thomas Fahringer
tf@dps.uibk.ac.at

1 University of Innsbruck, Technikerstraße 21a, 6020 Innsbruck, Austria

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 deriving a set of compile-time parameters for a parallel runtime system.
- A set of six 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 [13], targeting five runtime parameters which affect execution time and memory consumption.
- An evaluation of our prototype implementation on 12 task-parallel programs on a shared-memory parallel system with up to 64 hardware threads.
- An empirical comparison of our results using the optimizations presented here with four state-of-the-art task-parallel runtime systems, to put these findings into a broader context.

This paper improves upon and extends work previously presented by the authors [26], adding an additional target optimization parameter, two more analysis algorithms, and an in-depth state-of-the-art comparison.

The remainder of this paper is structured as follows. We first provide some measurements illustrating the potential improvements possible by optimal parameter selection 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

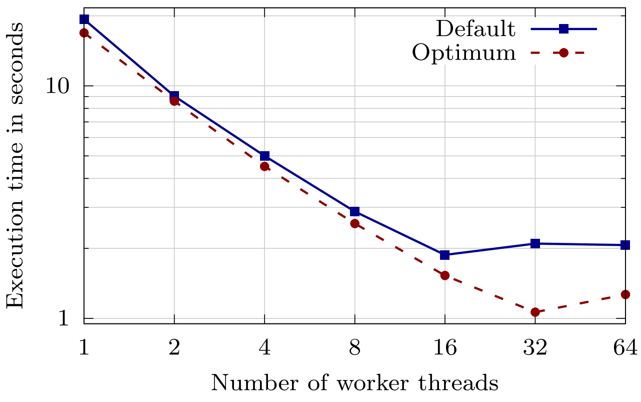

Figure 1 Execution time of the *Strassen* benchmark.

implementation are discussed in Section 4. An overview of related work is provided in Section 5 before concluding the paper.

2 Motivation

Before committing 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 – spanning a set of work queue insertion policies and sizes, as well as settings for several buffers and initial stack dimension – 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 [7], 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 two of the runtime parameters we identified as candidate for static tuning primarily influence memory consumption, Fig. 2 depicts a similar comparison for this aspect of performance. The relative advantage is lower, but still significant, reaching 36% in memory usage reduction at 64 threads.

Across the full set of benchmarks described in Section 4.2, *Strassen* is an average example in terms of

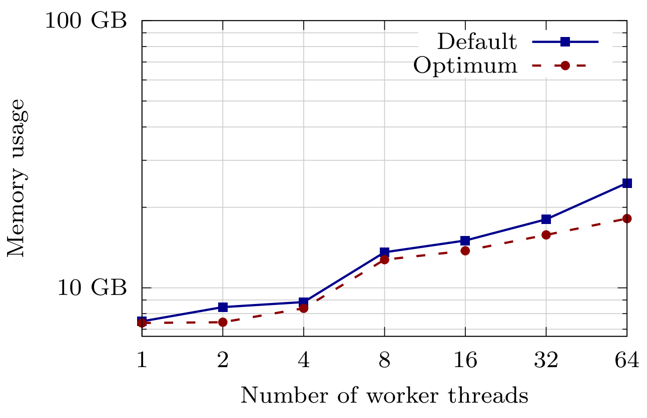

Figure 2 Memory consumption of the *Strassen* benchmark.

optimization potential with optimal static parameter selection. As such, a maximum improvement by a factor of 1.97 in execution time and a reduction of 36% in memory consumption is a very encouraging sign for our approach.

3 Method

An overview of our proposed method is provided in Fig. 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 as individual *Task IR* ②. Each task code fragment is then processed by the specialized analyses presented in this work ③. The results of these analyses are aggregated as required, 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 allow for work ranges with runtime-directed splitting, binary multi-versioning [23], and annotation of meta-data by the compiler [24].

This runtime system has been previously demonstrated [23] 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.

All investigated parameter values along with their defaults (in bold face) are depicted by Table 1.

Figure 3 Overview of our method.

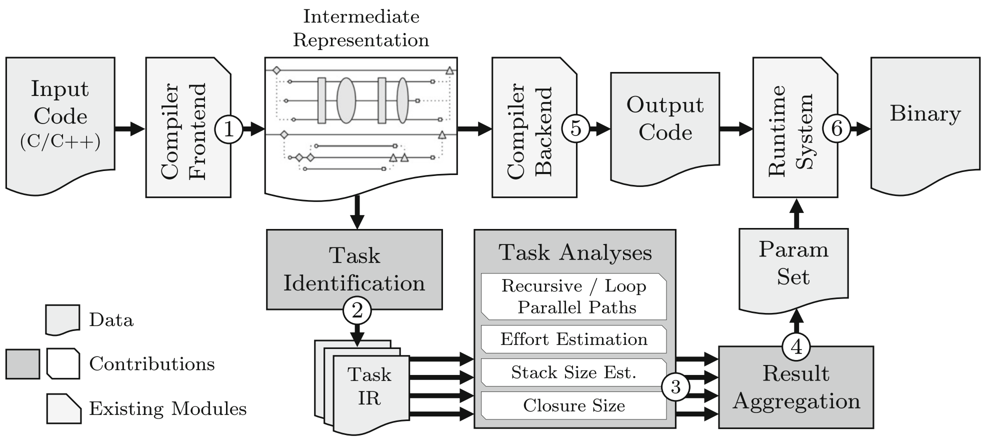

Table 1 Tuned parameters with all evaluated values (defaults marked in bold).

Runtime Parameter	Evaluated Values
Queue Policy	**Self Push Back/Other Pop Back**, Self Push Front/Other Pop Back, Self Push Back/Other Pop Front, Self Push Front/Other Pop Front
Queue Size	4, 8, **16**, 32, 64, 128
Event Table Buckets	**97**, 1021, 64567, 256019
Parameter Buffer Size	32, 64, **128**, 256, 512, 1024
Stack Size	16 kB, 32 kB, 64 kB, 128 kB, 256 kB, 512 kB, 1 MB, 2 MB, 4 MB, **8 MB**

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 Fig. 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. Note that while a worker always looks at the front of its own queue for tasks, making this feature a configurable parameter would only result in four additional policies which mirror the behavior of the existing ones. We have confirmed this idempotence across a large set of benchmarks before eliminating this parameter from further considerations.

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 [15], 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. The underlying assumption is that in case the deque is full, enough parallelism is available in the system, and the overhead of generating more steal-able tasks can be avoided.

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.

Event Table Buckets For use cases which unavoidably require some type of global knowledge or bookkeeping,

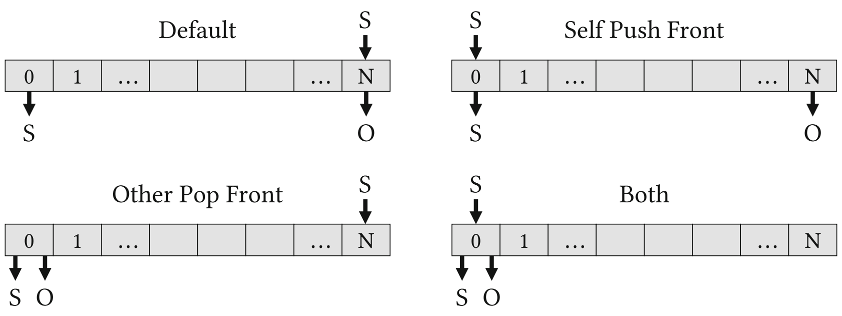

Figure 4 Behavior of available queue policies.

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 some larger prime numbers.

The number of buckets in the event hash table needs to 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 $O(N)$.

Parameter Buffer Size Whenever a task is generated by an executing program, its set of parameters and captured closure need to be stored for future use, potentially on another worker. Crucially, this data might need to outlive the current stack frame, and therefore requires heap storage. In order to optimize this process for small tasks, the runtime system allocates a fixed-size buffer area within each work item data structure. If the required data fits within that buffer, it is used directly. If not, an additional heap memory allocation is required every time a task is spawned.

If the required closure parameter data size for most common tasks in a program can be determined statically, a fixed buffer size accommodating these requirements can be chosen exactly, eliminating the dynamic allocation overhead without incurring additional memory requirements for over-provisioning.

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 tasking 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.

3.2 Compiler Analysis

The 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.

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 [12] 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 [11]. 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 2, 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.

Table 2 INSPIRE IR 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 (non-recursively).

– 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 analyses on INSPIRE IR are inherently whole-program and inter-procedural. Task execution generally requires capturing of context data and passing executable code as a parameter to a higher-order function, so local analysis rarely provides 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 $\mathcal{C}^n$ 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. For example, $n_{(i,j)}$ refers to the jth child node of the ith child of n.

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 given 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 its associated *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 of a leaf addresses (with $|\mathcal{C}^a| = 0$) reachable from node n.

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

closure_of(n) Computes the closure of node n and returns the list of types of all variables captured in it.

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 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 of our method provided in Fig. 3. The input to this step is a full program in INSPIRE IR, and its outputs are root IR nodes of the task code fragments identified.

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:     $f \leftarrow t_{(0,1)}$
 6:     for all $n \in$ reverse_sequence($t$) do
 7:         if $\exists A \mid n =$ call_of($f, A$) $\wedge \neg$is_builtin($f$)
 8:             $t \leftarrow f$
 9:             break
10:         end if
11:     end for
12:     $T' \leftarrow T' \cup \{t\}$
13: end for
14: return $T'$
```

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 IR.

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*.

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 sub-tasks, 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 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:     $l' \leftarrow \bot$
 4:     $p \leftarrow \bot$
 5:     $c \leftarrow \bot$
 6:     for all $n \in$ reverse_sequence($a$) do
 7:         if $\exists l \mid n =$ call_of_ref($l$)
 8:             $l' \leftarrow l$
 9:             $c \leftarrow n$
10:         else if $c \wedge \exists A \mid n =$ call_of($parallel$, $A$)
11:             $p \leftarrow \top$
12:         else if $c \wedge p \wedge n =$ def_of($l'$)
13:             $P \leftarrow P \cup (n, c)$
14:             break
15:         end if
16:     end for
17: end for
18: return $P$
```

Algorithm 2 determines the set of recursive parallel paths P within a given task invocation context. It traverses the address node sequence of each possible leaf address

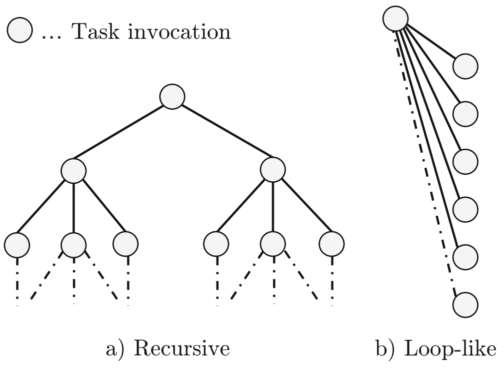

Figure 5 Fundamental parallel program structures.

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. Note that a call by reference always indicates a recursive invocation in the INSPIRE-IR, as its structural design would include the full implementation instead of a reference at the call site were it not recursive. The definition of the recursive callee always occurs at the call site of some function in the same (mutually) recursive set of functions.

Figure 6 shows a simplified example of an INSPIRE IR address tree for a task-parallel program, as it would be generated from the C++/OpenMP program outlined in Listing 1. 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

```
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-IR address tree structure.

```
void foo() { Ⓐ
  ...
  if(...) {
    ...
    foo(); Ⓑ
    ...
  } else {
    ...
    #pragma omp task Ⓒ
    foo(); Ⓓ
  }
  ...
  #pragma omp taskwait
}

int main() {
  ...
  foo();
}
```

Listing 1 Example C++ code snippet.

invocation 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.

Algorithm 3 Determine Loop-like Parallelism

```
     t        root node of the task context
 1: $L \leftarrow 0$
 2: for all $a \in$ all_leaf_addresses$(t)$ do
 3:     $p \leftarrow \bot$
 4:     for all $n \in$ reverse_sequence$(a)$ do
 5:         if $\exists A \mid n =$ call_of$(parallel, A)$
 6:             $p \leftarrow \top$
 7:         else if $p \wedge (\exists i, b, h \mid n =$ loop$(i, b, h))$
 8:             $L \leftarrow L + 1$
 9:             break
10:         end if
11:     end for
12: end for
13: return $L$
```

Algorithm 3 is the counterpart to Algorithm 2 for loop-based parallel structures. It follows the same overall pattern of traversing the reverse sequence of each leaf address in the given task context, but the logic is somewhat simpler as it does not have to detect and distinguish individual recursion nests. If any loop construct is found "above" a parallel invocation – that is, prior to it in the reverse node sequence – that parallel invocation is counted as loop-like.

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

Algorithm 4 performs a static *effort estimation* on an arbitrary INSPIRE IR node n. By default, it simply traverses all child nodes (line 18). 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.

Algorithm 4 Effort Estimation

```
     B        effort mapping function for built-ins
 1: function EFFORT$(n)$
 2:     $e \leftarrow 0$
 3:     if $\exists f, A \mid n =$ call_of$(f, A)$
 4:         for all $\alpha \in A$ do
 5:             $e \leftarrow e +$ EFFORT$(\alpha)$
 6:         end for
 7:         if n = call_of_ref$(f)$
 8:             return $e$
 9:         end if
10:         if is_builtin$(f)$
11:             return $e + B(f)$
12:         end if
13:         return $e +$ EFFORT$(f)$
14:     end if
15:     if $\exists i, b, h \mid n =$ loop$(i, b, h)$
16:         return $i * ($EFFORT$(b) +$ EFFORT$(h))$
17:     end if
18:     for all $c \in \mathcal{C}^n$ do
19:         $e \leftarrow e +$ EFFORT$(c)$
20:     end for
21:     return $e$
22: end function
```

3.2.6 Stack Size Estimation

This first memory usage 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 5 illustrates, stack size estimation for a given task context can be expressed quite succinctly due to the properties of INSPIRE-IR. 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, as only a single path in the execution tree can be traversed at a time. Thanks to the IR structure, this simple principle accurately covers various cases such as function call arguments, compound statements, and all types of control flow.

Algorithm 5 Stack Size Estimation

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

```
 1: function STACK_SIZE($n$)
 2:     if $\exists\tau, i \mid n = \text{declaration}(\tau, i)$
 3:         $s \leftarrow S(\tau)$
 4:         return $(s, s + \text{STACK\_SIZE}(i))$
 5:     end if
 6:     $(p, q) \leftarrow (0, 0)$
 7:     for all $c \in \mathcal{C}^n$ do
 8:         $(p', q') \leftarrow \text{STACK\_SIZE}(c)$
 9:         $p \leftarrow p + p'$
10:         $q \leftarrow max(q, q')$
11:     end for
12:     if $\exists l \mid n = \text{call\_of\_ref}(l)$
13:         if $\text{def\_of}(l) \neq t \wedge l \notin V$
14:             $V \leftarrow V \cup l$
15:             $(p', q') \leftarrow \text{STACK\_SIZE}(\text{def\_of}(l))$
16:             return $(p + p' * \phi, q + q' * \phi)$
17:         end if
18:     end if
19:     return $(p, q)$
20: end function
```

3.2.7 Closure Size Computation

The final task analysis we developed is designed to enable the static selection of an ideal parameter buffer size (as described in Section 3.1.2). For this purpose, the total size of all variables captured in the closure of the task invocation needs to be computed. In case several parallel task invocations exist within a single task context, the maximum needs to be reported. Algorithm 6 performs the necessary analysis, reusing the same type size mapping S leveraged by Algorithm 5.

Note that the $closure\_of(i)$ operation, while somewhat complex, in fact already needs to be performed by the compiler over the course of translating any given task-parallel program, since this closure must be encoded in the output program. As such, using this existing information for the additional purpose of encoding a suitable parameter buffer size is a very cheap operation a compile time ($O(p * v)$ with p parallel invocations and an upper bound of v variables captured per closure).

Algorithm 6 Closure Size Computation

t	root lambda of the task context
S	type size mapping

```
 1: $P \leftarrow \text{all\_calls\_of}(t, parallel)$
 2: $I \leftarrow \{parallel(job(r, f)) \in P \mid r = [1, 1]\}$
 3: $R \leftarrow 0$
 4: for all $i \in I$ do
 5:     $r \leftarrow 0$
 6:     for all $\tau \in \text{closure\_of}(i)$ do
 7:         $r \leftarrow r + S(\tau)$
 8:     end for
 9:     $R \leftarrow \max\{R, r\}$
10: end for
11: return $R$
```

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 and Closure Size

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

3.3.4 Deriving Parameter Values

While the one-to-one mapping from the stack and closure size analysis results to the actual runtime parameters 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 4 in the evaluation section.

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

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

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

For the queue policy parameter, the decision tree chooses the "Self Push Front" (PF) strategy 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 with four Intel Xeon E5-4650 processors, each offering 8 cores (16 hardware threads), which are 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 worker 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 3 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 [7], while the QAP2 benchmark was introduced in the Inncabs [22] suite. Both of these publications describe the involved benchmarks in some detail. The structure (loop-like, recursive balanced or recursive unbalanced) and granularity indicators in Table 3 are sourced from these publications, and based on human judgment and measurements of each code.

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 4 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 3 reveals interesting correlations:

– The only benchmarks with $\rho = 0$ are categorized as loop-like, confirming this result.

Table 3 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
Fibonacci	–	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

– 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 4 lists the results of our granularity analysis (Algorithm 4). 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.

In terms of memory analysis, the *Stack* column in Table 4 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.

Finally, the *Closure* column indicates the task closure size computed by our analysis. We have verified these results by comparing them with the final closure structure size in each generated output code, and all of them are accurate. The most interesting result in this column is the relatively large size for *Floorplan*, which is a result of a fixed-size integer array being captured by value. As the performance evaluation will demonstrate, this large value results in a more significant impact of the parameter buffer size optimization for this particular benchmark.

Table 4 Benchmark properties (analyses).

Benchmark	ρ	λ	Effort	Stack	Closure
Alignment	0	1	2.6 T	8 M	56 B
Delannoy	3	0	131.0	16 k	32 B
FFT	27	1	970.0 M	256 k	56 B
Fibonacci	2	0	52.0	16 k	24 B
Floorplan	1	1	39.0 G	2 M	576 B
Health	1	1	33.0 G	32 k	24 B
NQueens	1	1	601.0 M	2 M	40 B
QAP2	1	1	13.0 M	16 k	48 B
Sort	6	0	2.4 G	32 k	56 B
SparseLU	0	3	3.3 P	16 k	40 B
Strassen	7	0	282.0 G	256 k	56 B
UTS	1	1	12.0 T	2 M	40 B

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 [23]. 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.
- 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 evaluated 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 generally appear to be tuned for smaller shared-memory systems.

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 8 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.

To round out our look at execution time optimization, Table 5 lists the default time (*Def.*), the optimium time (*Opt.* – derived from exhaustive search), and the resulting time of our analysis and classification method (*Class.*) for each benchmark. Finally, the % column lists the ratio of the improvement of the absolute optimum over the default which was achieved by our method. Note that for most realistic benchmarks, this lies between 80% and

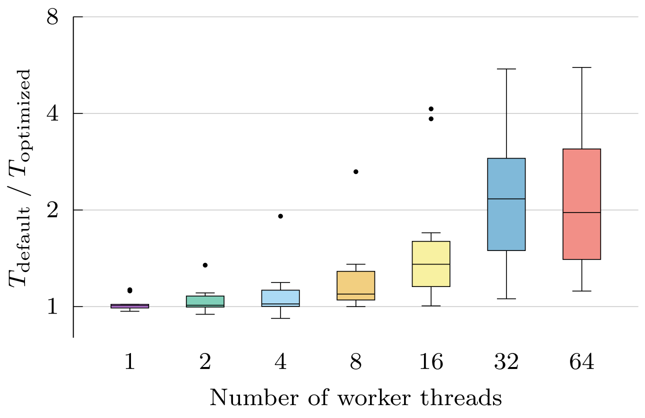

Figure 7 Overall execution time comparison.

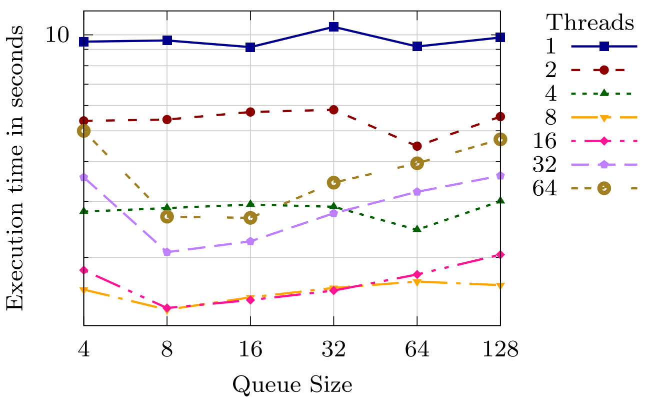

Figure 8 QAP2 execution times with varying queue size.

Table 5 Achieved speedup vs. optimum.

Benchmark	Def.	Opt.	Class.	%
Alignment	0.49	0.38	0.39	98.9%
Delannoy	0.61	0.20	0.34	59.6%
FFT	0.94	0.40	0.41	98.0%
Fibonacci	0.58	0.14	0.22	65.0%
Floorplan	2.25	0.89	0.89	99.7%
Health	2.24	1.50	1.73	86.7%
NQueens	2.76	1.14	1.42	80.7%
QAP2	1.47	0.41	0.59	69.9%
Sort	1.63	1.04	1.04	100.0%
SparseLU	0.51	0.45	0.46	99.8%
Strassen	1.88	1.06	1.24	85.9%
UTS	0.79	0.56	0.66	84.1%

100%. Over-fitting our method to the very fine-granular benchmarks would produce worse results for realistic programs.

4.4.2 Memory Consumption

Since two of the parameters we optimize primarily affect memory consumption, we also evaluated this aspect of runtime system performance. Figure 9 provides this overview, using the same methodology as employed for Fig. 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.

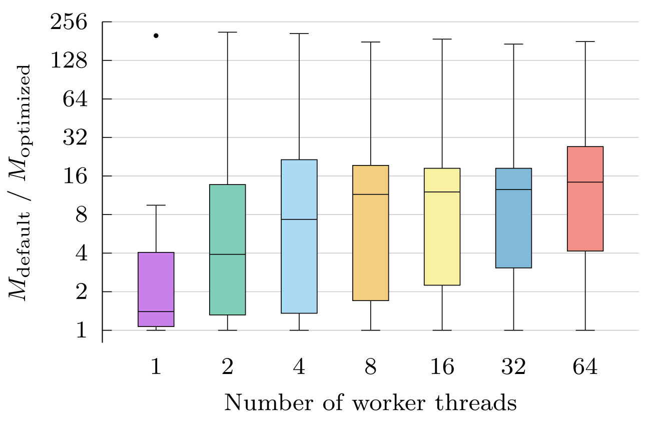

Figure 9 Overall memory consumption comparison.

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 these 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 *Fibonacci*, 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 [9] would not benefit as dramatically from this optimization.

An interesting observation can be made by looking at the data in more detail: the memory optimizations we perform in this work can also result in performance improvements in some very specific scenarios. In particular, the *Floorplan* benchmark requires a larger parameter buffer size than the default, and using this knowledge enables a significant performance improvement of up to 46%, which can be seen in Fig. 10. Note that the difference once again becomes more pronounced at larger degrees of parallelism – with more threads, more tasks are invoked, and with an insufficient parameter buffer size each such invocation requires an additional heap memory allocation.

4.5 State-of-the-art Comparison

While our evaluation so far demonstrates that applying static analyses in order to optimize task-parallel runtime

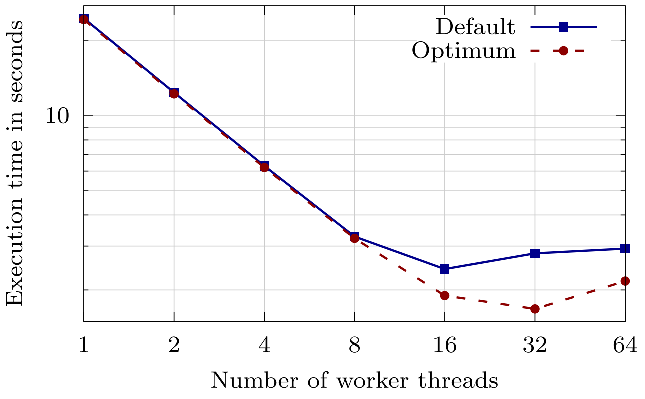

Figure 10 Floorplan execution times with optimal parameter buffer size.

system parameter selection provides significant performance improvements within the Insieme runtime system, it is not yet clear whether these are merely improvements over a low baseline.

This section is designed to provide a more holistic assessment of the quality of our approach, and to illustrate that the comparative basis chosen for our performance evaluation is meaningful. To this end, we will compare the absolute performance achieved by of our approach to four state-of-the-art and widely-used task-parallel systems in this section.

OpenMP is the industry standard for shared-memory parallelism, and supports task parallelism since version 3.0 [3]. This task support has since been widely deployed and studied [19]. For our evaluation, we use the GCC OpenMP implementation, *libGOMP* [17].

Cilk+ continues the development of the Cilk [5] language, which is considered the first widely-deployed high-performance task parallel system. It pioneered many techniques implemented in modern runtime systems [9]. Again, we use the implementation included in GCC.

Intel TBB [20] is a highly optimized parallel programming library widely used in industrial applications. For this evaluation, we used version 2017 Update 7 (20170604oss) of the library.

HPX [14] is a general purpose C++ runtime system for parallel and distributed applications of any scale. In our comparison, we employed the most recent git revision 62b12248ce of the library.

The backend compiler for all of these technologies was the same as the one used to compile the output programs of our system, GCC 5.1.0. All systems were configured according to the optimal settings in their user guides, e.g. HPX uses *tcmalloc* [10] rather than the default system allocator.

As performing this comparison required us to write versions of the benchmarks for some of the technologies in cases where they did not already exist, we focused on a subset of three benchmarks, representing three very distinct task granularities. Figures 11, 12 and 13 show the comparison for the *Fibonacci*, *Strassen* and *SparseLU* benchmark respectively.

As previously described, *Fibonacci* is a very unrealistic workload designed to test how an implementation reacts to tasks of minimal size. Both OpenMP and HPX are not designed to deal with this type of workload, and require more than 100 seconds (our cut-off point) at most thread counts. Intel TBB has some fixed overhead, but scales well up to 32 cores. Cilk+ is by far the most proficient at this particular scenario, scaling well without overhead. Our optimized version keeps up with Cilk+ on a single CPU, but fails to scale beyond that. This is due to work item synchronization inefficiencies in the underlying runtime, and unrelated to the compiler-based optimizations in this work.

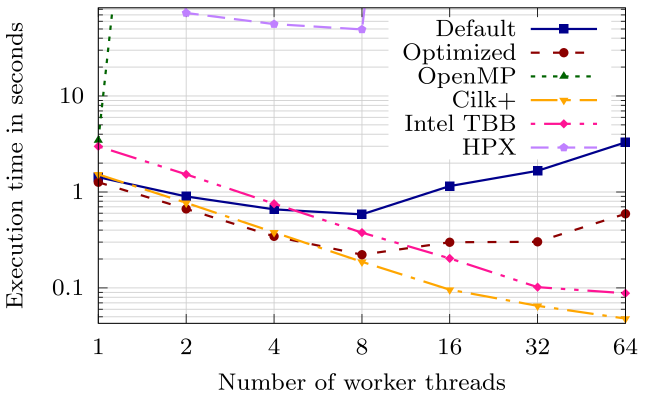

Figure 11 Performance charts for the *Fibonacci* benchmark.

Strassen is a much more representative test case, as it features tasks of medium granularity as they might be found in real-world codes. All implementations scale relatively well up to 8 cores. OpenMP, Cilk+ and TBB follow the same overall pattern, scaling up to 16 cores and then dropping off, with TBB offering higher absolute performance. The default Insieme system also follows this pattern. HPX shows a regression in performance when crossing the NUMA boundary, but then scales well to 32 cores. Our optimized version scales to 32 cores, primarily due to the scheduling policy being adjusted in a way that allows for good cache reuse.

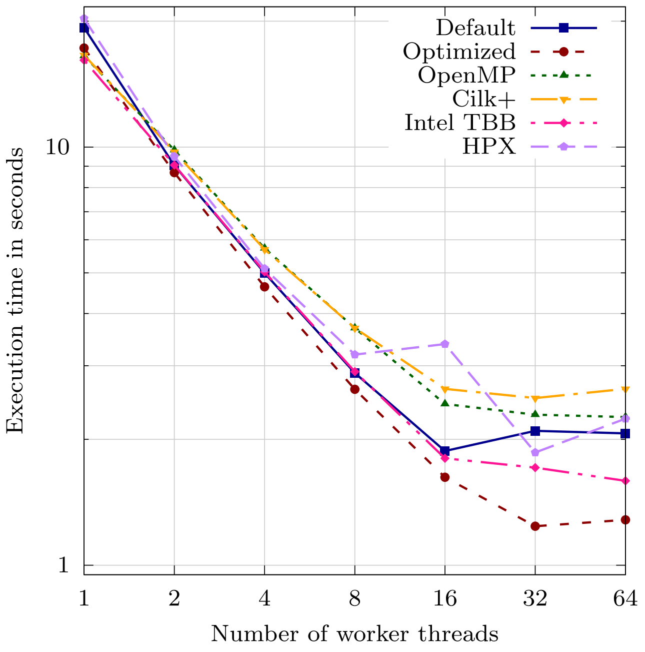

Figure 12 Performance charts for the *Strassen* benchmark.

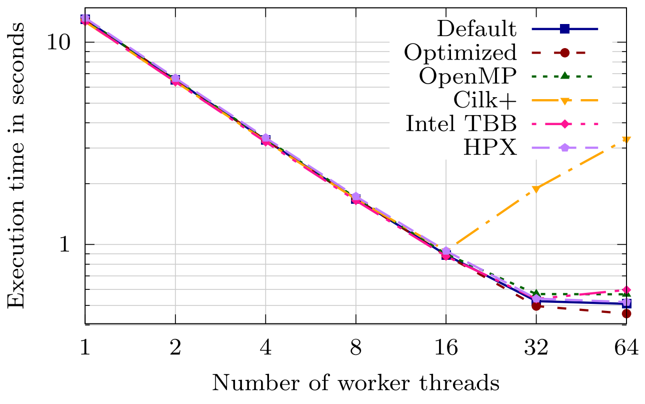

Figure 13 Performance charts for the *SparseLU* benchmark.

Finally, *SparseLU* features very large-grained, uniform tasks, and as expected the impact of the runtime system on performance is smaller. Except for Cilk+ – which fails to scale beyond two sockets – all other systems scale well up to 32 cores.

Summarizing these results, Fig. 14 compares the best performance achievable using each of these technologies, at their respective optimal degree of parallelism. In the pathological *Fibonacci* case, Cilk+ performs significantly better than any other competing technology.

In both of the more realistic applications, the Insieme default is already very competitive with the best competing technologies, and our optimization approach presented in this work results in significant further improvements. For *Strassen*, the best-performing competitor is Intel TBB, and our optimized solution is 28% faster.

In *SparseLU* all results are closer, with only Cilk+ showing a large difference due to its failure to scale on additional NUMA domains. Despite this smaller potential, our optimizations still result in a 14% performance improvement compared to HPX, which is the best-performing competing technology in this case.

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 optimization with a parallelism-specific compiler analysis component.

Tick and Zhong [27] propose a combined compile-time and run-time 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 [23], we leveraged a compiler component to control task granularity, but rather than providing estimates, granularity was actively adjusted by multiversioning of task functions.

A similar approach is followed by Baskaran et al. [4]. The automatic parallelization system *Pluto* can generate parallel OpenMP programs by analyzing and transforming sequential input codes. In order to improve load balancing on multi-core target platforms, the compiler generates functions which allow the runtime system to extract inter-tile data dependencies and thus improve scheduling decisions and reduce load-imbalance. The focus of their work is to improve the performance of loop-based auto-parallelized code generated from sequential input codes, while our approach works with parallel input programs and tries to improve runtime system performance based on static features of task-parallel programs.

Vuduc et al. [28] 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.

Another work which takes advantage of forwarding compiler analysis results to the runtime system is described by Sabeghi et al. [21]. Their focus is on improved task scheduling for reconfigurable processors. This is achieved by forwarding additional meta-data – specifically a so-called *configuration call graph*, which contains information about dependencies between tasks – along with the program binary to the runtime scheduler. This enables the scheduler component to take better decisions when configuring the hardware for task execution. Our work differs primarily in the targeted execution environment, and also in the specific analyses used. While Sabeghi et al. use the compiler-provided information to configure the hardware environment, our goal is to tune runtime system parameters for improved program execution on general purpose processors.

In the context of software distributed shared memory systems, Dwarkadas et al. [8] implement a combined compile-time and run-time method. The compiler component analyzes 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,

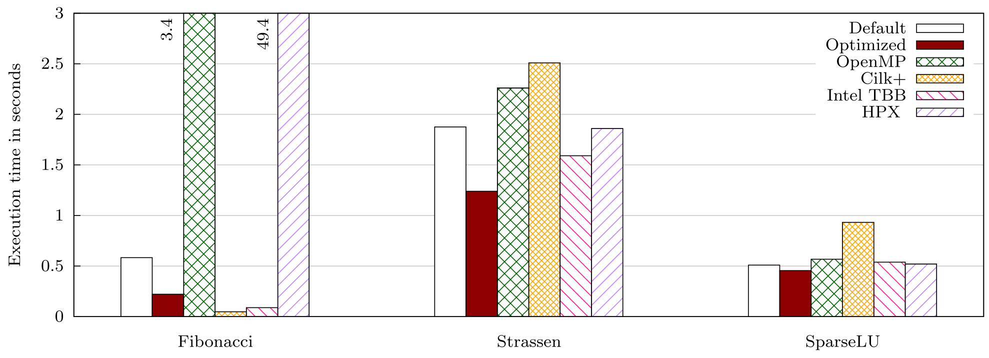

Figure 14 Performance comparison (at best thread count) of all evaluated technologies.

and thus reduce runtime overheads. Another approach combining a custom compiler component with a runtime library is described by Nikolopoulos et al. [16]. 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 [24] 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 primitives for task-based parallelism [25]. While an ad-hoc task classification procedure was implemented, 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 statically identifying tasks and performing a set of compiler analyses – specifically designed to classify and characterize their features – on each of them. 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.

An in-depth empirical comparison to existing task-parallel technologies indicates that, for realistic applications, our optimization approach can improve performance by up to 28% compared to the best-performing state-of-the-art systems.

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, particularly for large-scale NUMA systems. Finally, our current prototype mapping from analysis results to parameter settings can be replaced by a more sophisticated and automated approach.

Overall, we believe that our research indicates that there is significant untapped potential in static analysis for improving the performance of parallel runtime systems. Unlike most compiler analysis work, analyses designed for the purpose of guiding the non-functional behavior of runtime systems are not required to be fully accurate; inaccuracies will only result in potentially degraded performance rather than program failure, and even partially correct results might well allow for improved performance compared to a complete lack of static information. This property greatly extends the scope of feasible static analyses.

Acknowledgements Open access funding provided by University of Innsbruck and Medical University of Innsbruck. This project has received funding from the European Union's Horizon 2020 research and innovation program as part of the FETHPC AllScale project under grant agreement No 671603 as well as from the FWF Austrian Science Fund as part of project I 1523 "Energy-Aware Autotuning for Scientific Applications".

References

1. Asanovic, K., Bodik, R., Catanzaro, B.C., Gebis, J.J., Husbands, P., Keutzer, K., Patterson, D.A., Plishker, W.L., Shalf, J., Williams, S.W., et al. (2006). The landscape of parallel computing research: A view from Berkeley. Tech. rep., Technical Report UCB/EECS-2006-183, EECS Department University of California, Berkeley.
2. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.A. (2009). StarPU: a unified platform for task scheduling on heterogeneous multicore architectures.
3. Ayguadé, E., Copty, N., Duran, A., Hoeflinger, J., Lin, Y., Massaioli, F., Teruel, X., Unnikrishnan, P., Zhang, G. (2009). The design of openmp tasks. *IEEE Transactions on Parallel and Distributed Systems*, *20*(3), 404–418.
4. Baskaran, M.M., Vydyanathan, N., Bondhugula, U.K.R., Ramanujam, J., Rountev, A., Sadayappan, P. (2009). Compiler-assisted dynamic scheduling for effective parallelization of loop nests on multicore processors. In *ACM sigplan notices* (Vol. 44, pp. 219–228). ACM.
5. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y. (1996). Cilk: an efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, *37*(1), 55–69.
6. Chen, S., Gibbons, P.B., Kozuch, M., Liaskovitis, V., Ailamaki, A., Blelloch, G.E., Falsafi, B., Fix, L., Hardavellas, N., Mowry, T.C., Wilkerson, C. (2007). Scheduling threads for constructive cache sharing on CMPs. In *Proceedings of the nineteenth annual ACM symposium on parallel algorithms and architectures, SPAA '07* (pp. 105–115). New York: ACM. https://doi.org/10.1145/1248377.1248396.
7. Duran, A., Teruel, X., Ferrer, R., Martorell, X., Ayguade, E. (2009). Barcelona openMP tasks suite: a set of benchmarks targeting the exploitation of task parallelism in openMP. In *2009 international conference on parallel processing* (pp. 124–131). https://doi.org/10.1109/ICPP.2009.64.
8. Dwarkadas, S., Cox, A.L., Zwaenepoel, W. (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* (pp. 186–197). New York: ACM. https://doi.org/10.1145/237090.237181.
9. Frigo, M., Leiserson, C.E., Randall, K.H. (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* (pp. 212–223). New York: ACM. https://doi.org/10.1145/277650.277725.
10. Ghemawat, S., & Menage, P. (2009). Tcmalloc: Thread-caching malloc.
11. Jordan, H. (2014). Insieme: a compiler infrastructure for parallel programs. Ph.D. thesis, Ph D. dissertation, University of Innsbruck.
12. Jordan, H., Pellegrini, S., Thoman, P., Kofler, K., Fahringer, T. (2013). INSPIRE: the Insieme Parallel intermediate representation. In *Proceedings of the 22nd international conference on parallel architectures and compilation techniques, PACT '13* (pp. 7–18). Piscataway: IEEE Press.
13. Jordan, H., Thoman, P., Durillo, J.J., Pellegrini, S., Gschwandtner, P., Fahringer, T., Moritsch, H. (2012). A multi-objective auto-tuning framework for parallel codes. In *2012 international conference for high performance computing, networking, storage and analysis (SC)* (pp. 1–12). https://doi.org/10.1109/SC.2012.7.
14. Kaiser, H., Heller, T., Adelstein-Lelbach, B., Serio, A., Fey, D. (2014). Hpx: a task based programming model in a global address space. In *Proceedings of the 8th international conference on partitioned global address space programming models* (p. 6). ACM.
15. Mohr, E., Kranz, D.A., Halstead, R.H. (1991). Lazy task creation: a technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, *2*(3), 264–280. https://doi.org/10.1109/71.86103.
16. Nikolopoulos, D.S., Papatheodorou, T.S., Polychronopoulos, C.D., Labarta, J., Ayguadé, E. (2000). UPMLIB: a runtime system for tuning the memory performance of OpenMP programs on scalable shared-memory multiprocessors. In Dwarkadas, S. (Ed.) *Languages, compilers, and run-time systems for scalable computers: 5th international workshop, LCR 2000 Rochester, NY, USA, May 25–27, 2000 Selected Papers* (pp. 85–99). Berlin: Springer. https://doi.org/10.1007/3-540-40889-4_7.
17. Novillo, D. (2006). Openmp and automatic parallelization in gcc. In *Proceedings of the GCC developers summit*.
18. Olivier, S.L., Porterfield, A.K., Wheeler, K.B., Spiegel, M., Prins, J.F. (2012). OpenMP task scheduling strategies for multicore NUMA systems. *The International Journal of High Performance Computing Applications*, *26*(2), 110–124. https://doi.org/10.1177/1094342011434065.
19. Olivier, S.L., & Prins, J.F. (2010). Comparison of openmp 3.0 and other task parallel frameworks on unbalanced task graphs. *International Journal of Parallel Programming*, *38*(5), 341–360.
20. Reinders, J. (2007). *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly Media Inc.
21. Sabeghi, M., Sima, V.M., Bertels, K. (2009). Compiler assisted runtime task scheduling on a reconfigurable computer. In *International conference on field programmable logic and applications, 2009. FPL 2009* (pp. 44–50). IEEE.
22. Thoman, P., Gschwandtner, P., Fahringer, T. (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* (pp. 94–98). https://doi.org/10.1109/PDP.2015.33.
23. Thoman, P., Jordan, H., Fahringer, T. (2013). Adaptive granularity control in task parallel programs using multiversioning. In Wolf, F., Mohr, B., an Mey, D. (Eds.) *Euro-Par 2013 parallel processing: 19th international conference, Aachen, Germany, August 26-30, 2013. Proceedings* (pp. 164–177). Berlin: Springer. https://doi.org/10.1007/978-3-642-40047-6_19.
24. Thoman, P., Jordan, H., Pellegrini, S., Fahringer, T. (2012). *Automatic OpenMP loop scheduling: a combined compiler and runtime approach* (pp. 88–101). Berlin: Springer. https://doi.org/10.1007/978-3-642-30961-8_7.
25. Thoman, P., Moosbrugger, S., Fahringer, T. (2015). *Optimizing task parallelism with library-semantics-aware compilation* (pp. 237–249). Berlin: Springer. https://doi.org/10.1007/978-3-662-48096-0_19.

26. Thoman, P., Zangerl, P., Fahringer, T. (2017). Task-parallel runtime system optimization using static compiler analysis. In *Proceedings of the computing frontiers conference* (pp. 201–210). ACM.
27. Tick, E., & Zhong, X. (1993). A compile-time granularity analysis algorithm and its performance evaluation. *New Generation Computing*, *11*(3), 271. https://doi.org/10.1007/BF03037179.
28. Vuduc, R., Demmel, J.W., Bilmes, J.A. (2004). Statistical models for empirical search-based performance tuning. *The International Journal of High Performance Computing Applications*, *18*(1), 65–94. https://doi.org/10.1177/1094342004041293.

Peter Thoman is an Assistant Professor at the University of Innbruck, Austria, where he also obtained his PhD. He is a core developer and designer of the Insieme research compiler and parallel runtime system, and was involved in multiple national and international research projects in this capacity. His research interests include fine-grained task parallelism, compiler-supported optimizations, accelerator computing and API design for parallelism.

Peter Zangerl is a research assistant and PhD student at the University of Innsbruck, where he also received his Bachelor and Masters degree. He is an active developer of the Insieme research compiler and parallel runtime system. His main research interests are compilers, as well as parallel and distributed runtime systems.

Thomas Fahringer is a Professor of Computer Science at the University of Innsbruck. He is leading a research group in the area of distributed and parallel processing which develops the ASKALON system to support researchers worldwide in various fields of science and engineering to develop, analyse, optimize and run parallel and distributed scientific applications. Furthermore, he leads a research team that created the Insieme parallelizing and optimizing compiler for heterogeneous multicore parallel computers. Fahringer was involved in numerous national and international research projects including 10 EU funded projects. Fahringer has published 5 books, 35 journal and magazine articles and more than 160 reviewed conference papers including 3 best/distinguished IEEE/ACM papers.