Dependency-Based Automatic Parallelization of Java Applications

João Rafael, Ivo Correia, Alcides Fonseca, and Bruno Cabral

University of Coimbra, Portugal
{jprafael,icorreia}@student.dei.uc.pt, {amaf,bcabral}@dei.uc.pt

Abstract. There are billions of lines of sequential code inside nowadays software which do not benefit from the parallelism available in modern multicore architectures. Transforming legacy sequential code into a parallel version of the same programs is a complex and cumbersome task. Trying to perform such transformation automatically and without the intervention of a developer has been a striking research objective for a long time. This work proposes an elegant way of achieving such a goal. By targeting a task-based runtime which manages execution using a task dependency graph, we developed a translator for sequential JAVA code which generates a highly parallel version of the same program. The translation process interprets the AST nodes for signatures such as read-write access, execution-flow modifications, among others and generates a set of dependencies between executable tasks. This process has been applied to well known problems, such as the recursive Fibonacci and FFT algorithms, resulting in versions capable of maximizing resource usage. For the case of two CPU bounded applications we were able to obtain 10.97x and 9.0x speedup on a 12 core machine.

Keywords: Automatic programming, automatic parallelization, task-based runtime, symbolic analysis, recursive procedures

1 Introduction

Developing software capable of extracting the most out of a multicore machine usually requires the usage of threads or other language provided constructs for introducing parallelism [1, 2]. This process is often cumbersome and error prone, often leading to the occurrence of problems such as deadlocks and race conditions. Furthermore, as the code base increases it becomes increasingly harder to detect interferences between executing threads. Thus, one can understand why sequential legacy applications are still the most common kind and, in some cases, preferred as they provide a more reliable execution.

Automatic parallelization of existing software has been a prominent research subject [3]. Most available research focuses on the analysis and transformation of loops as the main source of parallelism [4, 5]. Other models have also been studied, such as the parallelization of recursive methods [6], and of sub-expressions in functional languages.

Our contribution is both a framework and a tool for performing the automatic parallelization of sequential JAVA code. Our solution extracts instruction *signatures* (read from memory, write to memory, control flow, etc.) from the application's AST and infers *data dependencies* between instructions. Using this information we create a set of tasks containing the same operations as the original version. The execution of these tasks is conducted by the Æminium Runtime which schedules the workload to all available cores using a work-stealing algorithm [7]. This approach supports a different number of processor cores by adjusting the number of worker threads and generated tasks, as long as there is enough latent parallelism in the program. With a simple runtime optimization, our experiments show a 9.0 speedup on a 12-core machine for the naive recursive Fibonacci implementation.

The remainder of this paper is organized as follows: in Section 2 we discuss the related work. Section 3 specifies the methodology used by the Æminium compiler throughout the entire process, from signature analysis to code generation. In Section 4 we conduct benchmarking tests and analyze the results. Finally, in Section 5 we present a summary of this paper's contributions and discuss future work.

2 Related Work

Extracting performance from a multicore processor requires the development of tailored, concurrent applications. A concurrent application, is composed by a collection of execution paths that may run in parallel. The definition of such paths can be done explicitly by the programmer with the aid of language supported constructs and libraries. An example of this approach is Cilk [8]. In the Cilk language, the programmer can introduce a division on the current execution path through the use of the *spawn* keyword. The opposite is achieved with the *sync* statement. When this statement is reached, the processor is forced to wait for all previously spawned tasks. A similar approach is used by OpenMP [9] where the programmer annotates a C/C++ program using pre-compiler directives to identify code apt for parallelism. Parallelism can also be hidden from the programmer. This is the case of paralleled libraries such as ArBB [8]. These libraries provide a less bug-prone design by offering a black-box implementation, where the programmer doesn't need to ponder concurrency issues but, still has no control over the amount of threads spawned for each library invocation.

For existing sequential program, these solutions require at least a partial modification of the application's source code. This may impose high rework costs, specially in the case of large applications, and may inadvertently result in the introduction of new bugs.

Automatic parallelization is an optimization technique commonly performed by compilers which target multicore architectures. By translating the original single threaded source code into a multi-threaded version of the same program, these compilers optimize resource usage and achieve lower execution times. Like all compiler optimizations, the semantics of the original source code must be preserved. As such, compilers must ensure the correct execution order between operations on multiple threads, taking into account their precedence in the original program.

One of the primary targets for automatic parallelization are loops. Numerical and scientific applications often contain loops consisting mostly of arithmetic operations. These loops provide a good source of parallelism due to the lack of complex control structures and can be parallelized with techniques such as *doall*, *doacross* and *dopipe* [10]. When dependencies between iterations are found the compiler may attempt to remove them by applying transformations such as variable privatization, loop distribution, skewing and reversal. These modifications are extensively described in [4].

Many algorithms however, are best implemented using a recursive definition as this is often the nature of the problem itself. The parallel resolution of each of the sub-problems has also been analyzed. In [11] this method is applied to the functional language LISP by the introduction of the **letpar** construct. This model can be used with success because the semantics of functional programming imply that there is no interference between sub-expressions. For non-functional languages, a technique known as thread-level speculation executes the operations optimistically assuming no interference. If such speculation is wrong, specialized hardware is used to rollback the faulty threads into a previous checkpoint [12]. In [13] recursion-based parallelism is applied to the JAVA language. In order to avoid interference between sub-expressions, a static analysis of read and write signatures is performed and the resulting data stored. At runtime, this information is used to check which methods can be executed in parallel by replacing the parameters with the actual variables in the stored method signatures. However, this runtime verification inadvertently introduces overhead. Our approach, on the other hand, does not resort to runtime support for dealing with this problem. By adding two new signatures, **merge** and **control**, we are able to solve this problem without a runtime penalty.

3 Methodology

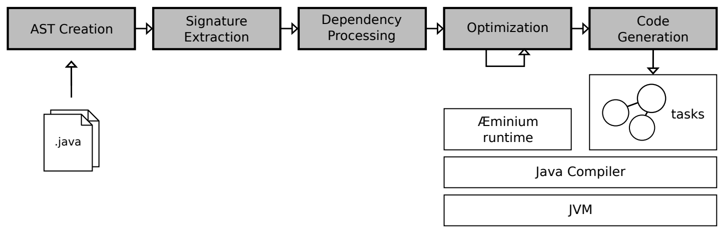


Fig. 1: Parallelization process used in the Æminium framework. Filled stages identify the source-to-source compilation described in this paper.

In order to extract parallelism from sequential programs, our framework decomposes a program into tasks to be scheduled at runtime using a work-stealing algorithm [7]. The entire process is depicted in figure 1. The first stage of the compilation process is the generation of the application's AST. This task is accomplished using Eclipse's JDT Core component which provides an API to read, manipulate and rewrite JAVA code. Each AST node is augmented with semantic information in the form of *signatures*. Signatures are a low-level description of what an instruction does, such as a read from a variable or a jump in the flow of the application. By transversing the AST in the same order as it would be executed, *data dependencies* and *control dependencies* are extracted and stored. Data dependencies identify mandatory precedence of operations due to concurrent access of the same variables whereas control dependencies indicate that the first operation designates whether or not the second executes. After this analysis, an optional phase of optimization takes place where redundant dependencies are removed and nodes are assigned into tasks. This optimization is repeated until no improvement is observed or a predefined threshold is achieved. Finally, this information is used to produce JAVA code for each task respecting the data and control dependencies in the program.

3.1 Signature Extraction

The analysis of the source program starts with the extraction of *signatures* for each node in the AST. Formally, signatures can be defined as predicates $S : \mathbb{A} \times \mathbb{D}^+ \rightarrow \{true, false\}$, where $\mathbb{A}$ is a set of AST nodes and $\mathbb{D}^+$ is a set of ordered datagroup tuples. A datagroup is a hierarchical abstraction of memory sections whose purpose is to facilitate static analysis of the application's memory (i.e.: function scopes, variables). A single datagroup, $\phi \in \mathbb{D}$, encompasses the entire application. This datagroup is broken down by classes, methods, fields, scopes, statements, expressions and variables forming sub-datagroups $\tau := (\phi, \varphi_0, \cdots, \varphi_n)$. As an example, a local variable `v` inside a method `m` of a class `c` is identified by $\tau_{var} := (\phi, \varphi_c, \varphi_m, \varphi_{var})$. An additional datagroup $\psi \in \mathbb{D}$ describes all memory sections unknown to the code submitted for analysis (i.e.: external libraries or native calls). Furthermore, two special datagroups τ_{this} and τ_{ret} are used as placeholders and are, in later stages, replaced by the actual datagroups that represent the object identified by the `this` keyword and the object returned by the containing method. A current limitation of the compiler, which we are currently working on, is the lack of array subdivision. As such, an entire array and each of its inner values are only modeled as a single datagroup.

Signatures are grouped into five categories. The **read**(α, τ) predicate indicates that operations in the sub-tree with root α can read memory belonging to datagroup τ. Likewise, **write**(α, τ) expresses that operations in the same sub-tree can write to datagroup τ. A more complex signature is **merge**(α, τ_a, τ_b). This signature implies that after operations in α, τ_a is accessible through τ_b. In other words, τ_b contains a reference to τ_a (i.e.: τ_b is an alias for τ_a), and an operation to one of these datagroups might access or modify the other. The fourth predicate, **control**(α, τ), denotes the possibility of operations in α to alter the execution flow of other operations inside the scope marked by the datagroup τ. The last predicate $\mathbf{call}_m(\alpha, \tau_o, \tau_r, \tau_{p_0}, \cdots, \tau_{p_n})$ is used as a placeholder for method calls; τ_o is the datagroup of the object that owns the method, τ_r is the datagroup where the return value is saved and τ_{p_x} is the datagroup for each of the invocation arguments. In program 1 the reader can observe an example of signatures extracted by the compiler. Also note that a $\mathbf{merge}(\alpha_{ret_1}, \tau_n, \tau_{ret})$ signature is detected as well. However, since `n` and `ret` are both integers this signature can be omitted.

```
int f(int n) {
    if (n < 2) { // read(α_cond, τ_n)
        return n; // write(α_ret1, τ_ret), control(α_ret1, τ_f)
    }
    return f(n - 1) + f(n - 2); // call_f(α_inv1, ∅, τ_inv1, τ_p0)
}
```

Program 1: The Fibonacci function with a excerpt of the extracted signatures indicated in comments. *inv* stands for function invocation, *ret* for return value, *f* for the current function f and *p0* is the first argument of the invocation.

Signature extraction is executed as a 2-pass mechanism. In the first pass, signatures for each node are collected and stored. In the second pass, the transitive closure is computed by iteratively adding each sub-node signature set with the one from its parent. In this step, $\mathbf{call}_m$ signatures are replaced with the full signature set of the corresponding method. The set is trimmed down by ignoring irrelevant signatures such as modifications to local variables, and modified so that the signatures have meaning in the new context: (1) formal parameter datagroups are replaced by the argument datagroups τ_{p_x} (2) the τ_{this} datagroup is replaced by τ_o and (3) the τ_{ret} datagroup is replaced by τ_r. During this same step, **merge** signatures are also removed in a pessimistic manner by adding all the **read** and **write** signatures as required to preserve the same semantics.

Regarding external functions, the compiler assumes they read and write to the ψ datagroup (ensuring sequential execution). For a more realistic (and better performing) analysis, the programmer can explicitly indicate the signature set for these functions in a configuration file (e.g.: to indicate that `Math.cos(x)` only reads from its first parameter τ_{p_0} and writes to τ_{ret}).

3.2 Dependency processing

In a sequential program, operation ordering is used to ensure the desired behavior. Line ordering, operator precedence, and language specific constructs (i.e.: conditional branches, loops, etc.) define an execution order σ_t on the set of AST nodes. Our compiler starts by assigning each executable node to a separate æminium task. As such, the same total order can be applied to the set of tasks. Dependencies between tasks are used to define a partial order σ_p, obtained by an arbitrary relaxation of σ_t. The operator $\alpha \prec_x \beta$ is used to indicate precedence of α over β on the σ_x order. Therefor, when σ_x is the partial order of tasks σ_p,

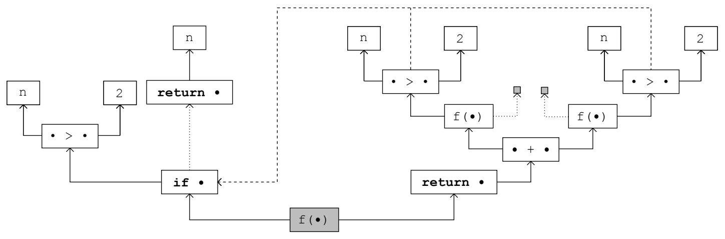


Fig. 2: Tasks generated for program 1 without optimization. Dotted arrows identify child scheduling. Solid arrows are used to represent strong dependencies while dashed arrows indicate weak dependencies. Filled tasks is the function root task

then $\alpha \prec_p \beta$ indicates the existence of a dependency from task β to task α. For the dependency set to be correct, any possible scheduling that satisfies σ_p has to have the exact same semantics has the one obtained with σ_t. The following rules are used to ensure this property:

1. A task that may read from a datagroup must wait for the termination of the last task that writes to it;

$$\frac{\alpha \prec_t \beta, \quad \forall \alpha, \beta \in \mathbb{A} \qquad \mathbf{write}(\alpha, \tau),\ \mathbf{read}(\beta, \tau)}{\therefore\ \alpha \prec_p \beta}$$

2. A task that may write to a datagroup must wait for the conclusion of all tasks that read from it since the last write;

$$\frac{\alpha \prec_t \beta \prec_t \gamma, \quad \forall \alpha, \beta, \gamma \in \mathbb{A} \qquad \mathbf{write}(\alpha, \tau),\ \mathbf{read}(\beta, \tau),\ \mathbf{write}(\gamma, \tau)}{\therefore\ \beta \prec_p \gamma}$$

   If two tasks may write to the same datagroup and there is no intermediary task that reads from it, then the latter task must wait for the former to complete; [1]

$$\frac{\alpha \prec_t \beta, \quad \forall \alpha, \beta \in \mathbb{A} \qquad \mathbf{write}(\alpha, \tau),\ \mathbf{write}(\beta, \tau)}{\therefore\ \alpha \prec_p \beta}$$

3. After a datagroup merge, the three previous restrictions must be ensured across all datagroups;

$$\frac{\begin{array}{c}\alpha \prec_t \beta \prec_t \gamma, \quad \forall \alpha, \beta, \gamma \in \mathbb{A} \\ \mathbf{write}(\alpha, \tau_a),\ \mathbf{merge}(\beta, \tau_a, \tau_b),\ \mathbf{read}(\gamma, \tau_b)\end{array}}{\therefore\ \alpha \prec_p \gamma}$$

[1] This rule applies when operations require both read and write access (such as the increment operator), or when tasks span more than a single operation.

$$\frac{\alpha \prec_t \beta \prec_t \gamma, \quad \forall \alpha, \beta, \gamma \in \mathbb{A} \\ \mathbf{read}(\alpha, \tau_a),\ \mathbf{merge}(\beta, \tau_a, \tau_b),\ \mathbf{write}(\gamma, \tau_b)}{\therefore \alpha \prec_p \gamma}$$

$$\frac{\alpha \prec_t \beta \prec_t \gamma, \quad \forall \alpha, \beta, \gamma \in \mathbb{A} \\ \mathbf{write}(\alpha, \tau_a),\ \mathbf{merge}(\beta, \tau_a, \tau_b),\ \mathbf{write}(\gamma, \tau_b)}{\therefore \alpha \prec_p \gamma}$$

4. Control signatures enforce dependencies from all the tasks of the scope whose execution path can be altered.

$$\frac{\alpha \prec_t \beta, \quad \forall \alpha \in \mathbb{A}, \beta \in \tau_{scope} \qquad \mathbf{control}(\alpha, \tau_{scope}),}{\therefore \alpha \prec_p \beta}$$

The set of dependencies is generated by transversing the AST tree using order σ_t and processing the signatures obtained in Section 3.1. A lookup table is used to store the set of tasks that access each datagroup. Furthermore, the information regarding which datagroups are merged is also stored. For each task, all of its signatures are parsed and dependencies are created to ensure properties 1 to 4. These data structures are updated dynamically to reflect the changes introduced. If a conditional jump is encountered, duplicates of the structures are created and each branch is analyzed independently. When the execution paths converge, both data structures are merged: 1) disparities between tasks are identified and replaced with the task that encloses the divergent paths. 2) datagroup merge sets are created by the pair-wise reunion of sets from both branches.

In Figure 2 we can observe the set of tasks generated from the AST for Program 1. Dotted arrows identify the optional *child* scheduling that occurs when the parent task is executing. Dashed arrows indicate a *weak dependency* relationship meaning the source task must wait for completion of the target task. Solid arrows denote a *strong dependency*, one where in addition to the property of weak dependency also signifies that the source task must create and schedule the target task before its execution.

3.3 Optimization

Optimization is an optional step present in most compilers. The Æminium java to java compiler, in its current shape, is capable of performing minor modifications to the generated code in order to minimize runtime overhead. This overhead is closely related to task granularity and the number of dependencies generated. As such, the optimization step focuses on these two properties. Nevertheless, on the post-compilation of the generated code, all the expected optimizations performed by the native JAVA compiler still occur.

This step solves the optimization problem using an iterative approach by finding small patterns that can be locally improved. The transformations described in the following sections are applied until no pattern is matched or a maximum number of optimizations is reached.

Redundant Dependency Removal The algorithm for identifying task dependencies performs an exhaustive identification of all the data and control dependencies between tasks. And, although these dependencies are fundamental for guaranteeing the correct execution of the parallel program, they are often redundant. The omission of such dependencies from the final code will not help to increase parallelism but will lower the runtime overhead. We identify two patterns of redundancy. The first instance follows directly from the transitivity relation of dependencies: given three tasks α, β and γ, if $\alpha \prec_p \beta$ and $\beta \prec_p \gamma$ then $\alpha \prec_p \gamma$. If the former is present it can be omitted from the dependencies set. The second instance takes into account the definition of child tasks. If $\alpha \prec_p \beta$, $\alpha \prec_p \gamma$ and, simultaneously, β is a child task of γ, then the former dependency can be omitted. This is possible because the runtime only moves a task to the `COMPLETED` state when it and all it's children tasks have finished.

Task Aggregation The first pass is to create one task per each node of the AST. However, the execution of a parallel program with one task for each AST node is several times slower than the sequential program, which makes task aggregation mandatory. By coarsening the tasks, we are able to lower the scheduling overhead and the memory usage. This optimization step attempts to reduce the number of generated tasks by merging the code of several tasks together in one task. The **aggregate**(α, β) operation has the following semantics: given two tasks $\alpha, \beta \in \mathbb{A}$, such that α is a strong dependency of β, we merge α into β by transferring all the dependencies of α into β, and placing the instructions of α before the instructions of β or a place of equal execution semantics (such as the right-hand side of an assignment expression).

Given that the code inside each task executes sequentially, by over-aggregating tasks the parallelism of the program is reduced. As such, we identify two types of task aggregation. *Soft aggregation* reduces tasks without hindering parallelism: if task β depends on α, and there is no other task γ that also depends on α, then α can be merged into β without loss of parallelism.

$$soft \triangleq \alpha \prec_p \beta \wedge \nexists \alpha \prec_p \gamma \Rightarrow \textbf{aggregate}(\alpha, \beta) \quad \alpha, \beta, \gamma \in \mathbb{A}$$

Hard aggregation on the other hand attempts to merge tasks even in other scenarios, such as lightweight arithmetic operations. Currently the optimizer aggregates all expressions with the exception of method invocations (including constructor calls). Also, statements where execution must be sequential (e.g.: the `then` block of an `if` statement) and their aggregation does not violate dependency constraints are also aggregated. Optionally full sequentialization of cycles can also take place. Using this feature disables parallelization of loops, but generates a lower runtime memory footprint.

3.4 Code generation

The Æminium runtime executes and handles dependencies between `Task`'s. These objects contain information about their state, and their dependencies. The actual code executed by each task exists in a `execute()` method of a class that implements the `Body` interface. This factorization allows for reuse of the same body object for multiple tasks. Bodies are constructed with a single parameter: a reference to the parent body if it exists and `null` otherwise. This allows access to fields of upper tasks where local variables and method parameters will be stored. Inside the constructor of the body, its task is created by calling the `Aeminium.createTask()` function which receives the body as it first parameter. The second parameter defines a *hints* object used by the runtime to optimize scheduling. This functionality is not used by the compiler and the default value of `NO_HINTS` is used. Strong dependencies of the task are instantiated in the constructor of the task body. This operation must take place after the creation of the `task` object (since it must be available as the parent task when scheduling those dependencies), and before the schedule of the task itself (since those tasks will be used inside the task dependency list).

Methods In real-life applications, the same method is invoked many times in different places. This makes the already mentioned approach of accessing parent's fields unsatisfactory for translating method tasks as it would require replicating the same method based on it where it is invoked. Instead, in addition to the parent object, these tasks receive the invocation arguments as arguments to the constructor of the task body. However, this requires those values to be known when the task is created. Therefore, its instantiation must take place inside the `execute()` method of corresponding method invocation expressions, where the tasks that compute each argument have already completed. Nonetheless, method invocation expressions, as well as all other expressions, must save their value in a special field `ret` before they reach the `COMPLETED` state. In order to do so, the `return` task of the invoked method places the value in `ret` upon its own execution. Furthermore, as a consequence of having all values computed prior to the construction of a method task, it is possible to conduct a runtime optimization. By checking if enough parallelism is already achieved – by checking if enough tasks are queued and all threads are currently working – it is possible to invoke the sequential (original) method. This optimization allows us to almost entirely remove the overhead of the runtime once enough parallelism has been reached.

Loops Loop statements such as `while`, `for`, and `do...while` allow for multiple iterations to execute the same lines of code. However, the actual instructions may vary from iteration to iteration. Furthermore, the instructions on the first iteration must wait for instructions prior to the loop (e.g. a variable declaration) while subsequent instructions only need to wait for one on the previous iteration (last modification). To allow this duality of dependencies two trees of tasks are created for each loop. The former contains dependencies belonging to the first iteration while the latter includes dependencies associated with the following iterations. The parent task of this second tree contains a `previous` field that points to the preceding instance, and inside the `execute()` method creates another instance of itself. Sub-tasks make use of this field to reference tasks of the previous iteration for their dependency list.

4 Evaluation

To validate our approach we compiled three sample applications using the Æminium compiler and executed the resulting tasks in a machine with the following specification: 2 Intel®Xeon®Processor X5660 (6 cores each, with hyper-threading, forming a total of 24 threads) and 24 GB of RAM. The applications include the recursive implementation of the Fibonnaci program already mentioned in Section 3, an application to numerically approximate the integral of a function given an interval, and finally a simple implementation of the Fast Fourier Transform (FFT) on an array of 2^{22} random complex numbers. The FFT application requires the generation of an array of `Complex` objects. This step is not considered for the benchmark time as it requires sequential invocations to `Random.nextDouble()`. Also, in order to minimize runtime overhead of cycle scheduling the option to sequentialize loops (as described in 3.3) was used. Each experiment was repeated 30 times. The results are depicted in Table 1 and Figure 3.

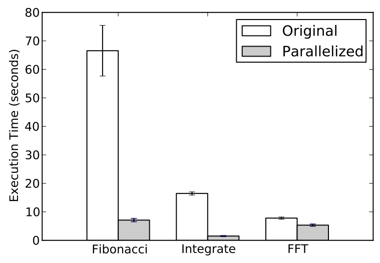
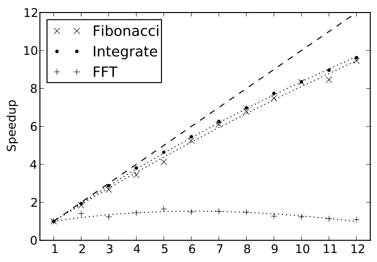


Fig. 3: Execution time before and after parallelization.



]

Fig. 4: Scalability benchmark for the three tests.

Application	Sequential	Parallel	Speedup
Fibonacci	55.56 (8.90) s	6.17 (v0.59) s	09.00
Integrate	16.46 (0.56) s	1.50 (0.19) s	10.97
FFT	07.80 (0.40) s	5.33 (0.40) s	01.46

Table 1: Measured average execution time (standard deviations) and speedups for the three benchmarks.

The first benchmark computes the 50th Fibonacci number. The sequential execution of this problem took on average 55.56 seconds to complete, while the parallel version only took 6.17 seconds. Although it consists of a 9.00x increase in performance ($p = 0.973$), it is well bellow the possible 12x (linear) speedup. The scalability test shown in Figure 4 indicates the cores/speedup relation. The dashed line is the desired linear speedup. The dotted lines identify the the least-squares method fitted to the Amdahl's law [14] with the exception of the third benchmark where an adjustment for linear ovearhead h was added.

The second benchmark computes the integral of the function $f(x) = x^3 + x$ in the interval $[-2101.0, 200.0]$ up to 10^{-14} precision. The behaviour of this test is similar to the previous, but with a slightly higher $p = 0.978$. The FFT benchmark shows the lowest speedup among the three executed benchmarks ($p_{amdahl} = 0.311$ or $p = 0.972, h = 0.746,$). It is also the one with highest memory usage. This suggests that memory bandwidth is the primary bottleneck of this particular implementation. In fact, this is the case for naïve FFT implementations as indicated in [15]. As a consequence, for larger arrays the speedup decreases as cache hits become less and less frequent due to false sharing.

5 Conclusion and Future Work

By targeting a task-based runtime, our framework is capable of automatically parallelizing a subset of existing java code. This solution provides respectable performance gains without human intervention. The compiler is able to detect parallelism available in loops, recursive method calls, statements and even expressions. The benchmarks executed show near-linear speedup for a selected set of CPU bounded applications.

Future work for this project includes testing the approach on a large suite of Java programs. In order to do that, the full set of Java instructions needs to be supported. This includes exception handling, reflection instructions (such as `instanceof`), class inheritance, interfaces, etc. The results on a large codebase would allows for a thorough analysis of the performance and optimizations required. One of the potential optimizations if the usage of a cost analysis approach to efficiently conduct hard aggregation of small tasks. This analysis should also take into account task reordering to further merge task chains. The current implementation of loop tasks introduces too much overhead to be of practical use, so the creation of tasks that work in blocks or strides should provide a better performing model.

Acknowledgments

This work would not have been possible without the contributions to the Aeminium language and runtime from Sven Stork, Paulo Marques and Jonathan Aldrich. This work was partially supported by the Portuguese Research Agency FCT, through CISUC (R&D Unit 326/97), the `CMU|Portugal` program (R&D Project Aeminium CMU-PT/SE/0038/2008), the iCIS project (CENTRO-07-ST24-FEDER-002003), co-financed by QREN, in the scope of the Mais Centro

Program and European Unions FEDER and by the COST framework, under Actions IC0804 and IC0906. The third author was also supported by the Portuguese National Foundation for Science and Technology (FCT) through a Doctoral Grant (SFRH/BD/84448/2012).

References

1. K Arnold, J Gosling, D.H.: The Java programming language. Addison Wesley Professional (2005)
2. Biema, M.v.: A survey of parallel programming constructs. In: Columbia University Computer Science Technical Reports. Department of Computer Science, Columbia University (1999)
3. Banerjee, U., Eigenmann, R., Nicolau, A., Padua, D.: Automatic program parallelization. Proceedings of the IEEE **81**(2) (feb 1993) 211 –243
4. Banerjee, U.: Loop Transformations for Restructuring Compilers: The Foundations. Springer (1993)
5. Feautrier, P.: Automatic parallelization in the polytope model. In Perrin, G.R., Darte, A., eds.: The Data Parallel Programming Model. Volume 1132 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (1996) 79–103 10.1007/3-540-61736-1_44.
6. Bik, A.J., Gannon, D.B.: Automatically exploiting implicit parallelism in java. Concurrency - Practice and Experience **9**(6) (1997) 579–619
7. Blumofe, R.D., Leiserson, C.E.: Scheduling multithreaded computations by work stealing. J. ACM **46**(5) (September 1999) 720–748
8. Randall, K.: Cilk: Efficient multithreaded computing. Technical report, Cambridge, MA, USA (1998)
9. Dagum, L., Menon, R.: Openmp: an industry standard api for shared-memory programming. Computational Science Engineering, IEEE **5**(1) (jan-mar 1998) 46 –55
10. Ottoni, G., Rangan, R., Stoler, A., August, D.: Automatic thread extraction with decoupled software pipelining. In: Microarchitecture, 2005. MICRO-38. Proceedings. 38th Annual IEEE/ACM International Symposium on. (nov. 2005) 12 pp.
11. Hogen, G., Kindler, A., Loogen, R.: Automatic parallelization of lazy functional programs. In: Proc. of 4th European Symposium on Programming, ESOP'92, LNCS 582:254-268, Springer-Verlag (1992) 254–268
12. Bhowmik, A., Franklin, M.: A general compiler framework for speculative multithreading. In: Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures. SPAA '02, New York, NY, USA, ACM (2002) 99–108
13. Chan, B., Abdelrahman, T.S.: Run-time support for the automatic parallelization of java programs. J. Supercomput. **28**(1) (April 2004) 91–117
14. Amdahl, G.M.: Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of the April 18-20, 1967, spring joint computer conference. AFIPS '67 (Spring), New York, NY, USA, ACM (1967) 483–485
15. da Silva, C.P., Cupertino, L.F., Chevitarese, D., Pacheco, M.A.C., Bentes, C.: Exploring data streaming to improve 3d fft implementation on multiple gpus. In: Computer Architecture and High Performance Computing Workshops (SBAC-PADW), 2010 22nd International Symposium on, IEEE (2010) 13–18