CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. (2011)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.1898

Resource analyses for parallel and distributed coordination

P. W. Trinder[1,*,†], M. I. Cole[2], K. Hammond[3], H-W. Loidl[1] and G. J. Michaelson[1]

[1]*School of Mathematical and Computer Sciences, Heriot-Watt University, Riccarton, Edinburgh EH14 4AS, UK*
[2]*School of Informatics, The University of Edinburgh, 10 Crichton Street, Edinburgh EH8 9AB, UK*
[3]*School of Computer Science, University of St. Andrews, St. Andrews, Fife KY16 9SX, UK*

SUMMARY

Predicting the resources that are consumed by a program component is crucial for many parallel or distributed systems. In this context, the main resources of interest are execution time, space and communication/synchronisation costs. There has recently been significant progress in resource analysis technology, notably in type-based analyses and abstract interpretation. At the same time, parallel and distributed computing are becoming increasingly important.

This paper synthesises progress in both areas to survey the state-of-the-art in resource analysis for parallel and distributed computing. We articulate a general model of resource analysis and describe parallel/distributed resource analysis together with the relationship to sequential analysis. We use three parallel or distributed resource analyses as examples and provide a critical evaluation of the analyses. We investigate why the chosen analysis is effective for each application and identify general principles governing why the resource analysis is effective. Copyright © 2011 John Wiley & Sons, Ltd.

Received 8 March 2011; Revised 30 September 2011; Accepted 3 October 2011

KEY WORDS: resource analysis; cost models; parallelism; distributed systems

1. INTRODUCTION

Accurately predicting the resources that will be consumed during program execution is important to a number of areas, including real-time systems, databases and parallelism. Although resource usage may be predicted in a black-box way, for example by profiling, this gives little information about worst-case bounds or untypical cases. It also fails to exploit information that could be derived by inspecting the program source and provides few, if any, guarantees about future behaviours. Early work on source-level resource analysis includes that of Cohen and Zuckerman, who translated Algol-60 programs into a symbolic form that conveyed cost information [1]; Wegbreit, who applied a similar approach to recursive Lisp programs in his metric system [2]; Ramshaw [3] and Wegbreit [4], who considered formal verification of cost specifications; and Hickey and Cohen [5], who focused on the theoretical foundations of a high-performance compiler that was capable of automatically generating functions describing average-case execution time. Notable early systems for automated complexity analysis are Complexa [6] and $\Lambda\Upsilon\Omega$ [7], which built on the metric system and extended it to cover average-case complexity analysis of algorithms. Recent theoretical advances include the development of powerful and flexible *type-based* approaches (e.g. [8–12]) that are capable of determining execution costs without expensive, data-dependent and possibly nonterminating symbolic execution.

Resource analysis has recently increased in importance because of the availability of improved technologies, such as type-and-effect systems [13, 14], combined with increasing real-world

*Correspondence to: P. W. Trinder, School of Mathematical and Computer Sciences, Heriot-Watt University, Riccarton, Edinburgh, UK.

†E-mail: P.W.Trinder@hw.ac.uk

emphasis on resource-constrained computing in areas such as embedded systems and cloud computing. Practical uses of resource analysis include providing guarantees for safety-critical systems, ensuring QoS for networks or embedded systems or providing information that can be used to make sensible decisions about the allocation of resources in for example database systems. Large-scale uses include the ASTREE system, which has been used to analyse the worst-case execution time (WCET) of the flight-control software for the Airbus A380 [15], and the use of the SPEED symbolic complexity analyser to analyse much of the .NET code base [16].

This paper surveys one particularly important application area, namely parallel or distributed computing. Parallel systems are gaining importance with the expansion of multicore and manycore machines. Similarly, distributed systems are gaining in importance with the widespread adoption of internet and cloud technologies. Allocating resources effectively is important to achieving good performance as the number of cores rises in current and future architectures.

A number of different resources are of interest for parallel and distributed systems, for example execution time for parallelism or power consumption in a network of low power devices such as sensors. Moreover, the resource information may be used for a number of purposes; for example, execution time predictions can be used to optimise the performance of parallel programs or to aid load management or scheduling within distributed systems. In summary, effective resource analyses provide information that enables better coordination of parallel and distributed computation.

1.1. Contributions made by the paper

We start by motivating resource prediction for parallel/distributed systems (Section 2). We then present and illustrate an informal general model of resource analysis (Section 3). The model codifies folklore, that is 'what is usually done'. It is, however, general with regard to the resources analysed and the bounds asserted. That is, the resources of interest may include execution time, memory usage, file handles or any other limited and quantifiable resource. Similarly, predictions may, for example, be formally verifiable worst-case bounds, probabilistic measures of worst-case or average-case behaviours, or simple estimations of resource usage.

The paper then makes the following contributions:

- It articulates the relationship between parallel/distributed analysis and sequential resource analysis. Section 4 describes how parallel/distributed resource analyses relate to sequential analyses and illustrates the relationship with simple parallel analyses. We show that parallel resource analysis is an instance of our general resource analysis model. We show how some parallel/distributed analyses take a two-level approach where the parallel/distributed analysis utilises the results of a sequential analysis and discuss the benefits of structuring parallel/distributed analyses in this way.
- It provides a recent survey of resource analyses for parallel and distributed computing. Sections 6–8 discuss the components of the general resource analysis model for the parallel/distributed context. Section 6 classifies cost models that underpin parallel/distributed resource analysis. The cost models are parameterised with execution costs on the target implementation, and Section 7 discusses how these costs can be obtained. Section 8 discusses the techniques that can be used to construct resource analyses, for example type inference or abstract interpretation. The survey is representative rather than exhaustive.
- It gives a critical evaluation of three representative parallel/distributed resource analyses. For each application in Section 9, we present the key elements of the resource analysis model, namely the cost model, the implementation model and the analysis. We present results showing that each analysis is *effective*, that is that the analysis improves the coordination. The applications utilise a range of cost models and analyses, and use the resource information for a range of coordination purposes including resource-safe execution, optimising parallel execution and enabling mobility.
- It identifies some general principles for effective parallel/distributed resource analysis. Section 9 investigates why the chosen analysis proves effective for each application, and Section 10.1 identifies principles governing when a resource analysis is effective.

DOI: 10.1002/cpe

The paper extends our previous work [17] by (i) addressing a general audience, (ii) introducing a general model of resource analysis, (iii) relating parallel/distributed resource analysis to sequential analysis, and (iv) providing a critical evaluation of three parallel/distributed resource analysis applications.

2. THE NEED FOR RESOURCE PREDICTIONS

This section motivates why resource predictions are necessary for parallel and distributed systems. A number of resources are of interest for parallel and distributed systems, for example execution time for parallelism or memory consumption in a network of embedded devices with limited random access memory. Of these, execution time is most commonly estimated, although architecture trends are making power consumption increasingly important. The resource information may be used for a number of purposes; for example, execution time predictions can be used to optimise the performance of parallel programs, to improve load management or scheduling in distributed systems, or to certify the maximum resources consumption by mobile code. We shall see examples of resource predictions being used for a variety of such purposes in Section 9.

As a concrete illustration, we consider the use of execution time predictions to optimise the performance of parallel programs.

2.1. Assessing potential parallelism

Before developing a parallel program, it is wise to assess whether good parallel performance can be achieved. We know that our ability to reduce the execution time using multiple processors is fundamentally limited and also that this limitation depends on the execution times of components of the program. If we suppose that the total sequential time T for the program comprises an inherently sequential portion S (e.g. to acquire initial data and integrate final results) and a potentially parallel portion P (e.g. to compute independent components of the final results), then from Amdahl's Law [18], the best parallel speedup achievable with N processors is as follows:

$$T = S + P$$
$$speedup = T/(S + P/N)$$

So speedup is bound by the inherently sequential portion and depends on the near-optimal deployment of the processors to share the potentially parallel portion. Thus, two important objectives in parallel programming are to minimise the inherently sequential portion of a program and to ensure that each of the N processors carries out a very similar proportion of the parallel portion of the program.

We must also communicate data and results between the processors and coordinate their activities, and doing so introduces time overheads that must be accounted for. These overheads may be either inherently sequential (e.g. to distribute initial data and receive final results) or potentially parallel (e.g. to transmit intermediate information between subsets of processors).

Hence, key issues for assessing potential parallelism are as follows:

- Which portions of a program are inherently sequential and which are potentially parallel?
- What are the sequential execution times of these portions?
- Which communication and coordination construct to introduce to best enable parallelism?
- What additional time overheads do these constructs bear?

2.2. Parallel programming

Despite the existence of mature methodologies for parallel programming (e.g. [19]), combinations of folklore, code inspection and profiling seem to prevail in common use. The folklore holds, for example, that

- communication is more costly than processing;
- parallelism is most beneficial where substantial activities may be carried out with minimal communication;

DOI: 10.1002/cpe

- activities become more substantial and require less communication when they are grouped; and
- repeated activities that can be separated into independent subactivities are good loci of potential parallelism;

That is, effort focuses on simultaneously maximising task granularity while minimising communication. Although communication costs are less on shared-memory systems, there is considerable evidence that excessive use of shared memory has a serious impact on performance. A typical approach to parallelising a sequential program is therefore as follows, and established methodologies have similar stages.

- *Inspect* the program and identify the top level computations, often loops.
- *Profile* the program and identify the most costly components.
- Hope that inspection and profiling coincide in identifying repeated, independent, high-cost components that can be distributed across multiple processors.

The next stage is to build a parallel version of the program by using appropriate constructs for communication and coordination depending on the target architecture. The parallel program is then profiled on a target platform. Often, the initial performance is disappointing, and a tuning cycle ensues of regrouping parallel activities to maximise task granularity while minimising communication and of reprofiling to establish whether acceptable performance has been reached.

2.3. Obtaining execution time predictions

Assessing how much potential parallelism there is in a program assumes that we have good mechanisms for determining how long software components take to execute sequentially. In our discussion of naïve parallel programming, we have referred to *profiling* as a way to determine this information, that is acquiring time measurements from an executing program. However, profiling has a number of deficiencies:

- It is *data dependent* – measurements usually cannot be generalised to cover all cases of interest.
- It cannot take account of rarely executed *control flows* that may have significant impacts on WCETs.
- It is expensive to obtain a significant body of profiling information, both in terms of computational time and often in terms of labour.
- The information that is obtained is platform specific and does not easily extrapolate to other similar systems.
- Profiling requires access to the execution platform and to a range of measurement tools, which is not always possible in for example embedded or cloud computing systems.

Although profiling can deliver basic information, it therefore has a number of limitations in the general case. What is needed in many situations is a way of cheaply and automatically obtaining information about the execution of software components before they are deployed in a specific situation and to do this in a way that covers all possible program execution paths. That is, we need a reliable *analysis* of the time (and other resources) used by the components of the program. This analysis could be either a *static* (or *compile-time*) analysis working automatically on the program source or a *design-time* analysis that is carried out manually by a programmer or algorithm designer.

3. AN INTRODUCTION TO RESOURCE ANALYSIS

This section introduces resource analysis, giving a general model of resource analysis, and an example sequential analysis. The aim of resource analysis is to take a program component and apply an analysis that

- will give an accurate picture of what the component costs on some implementation platform,
- takes considerably less time and effort than profiling the component when it is executing on the platform and
- can be straightforwardly applied to other implementation platforms.

DOI: 10.1002/cpe

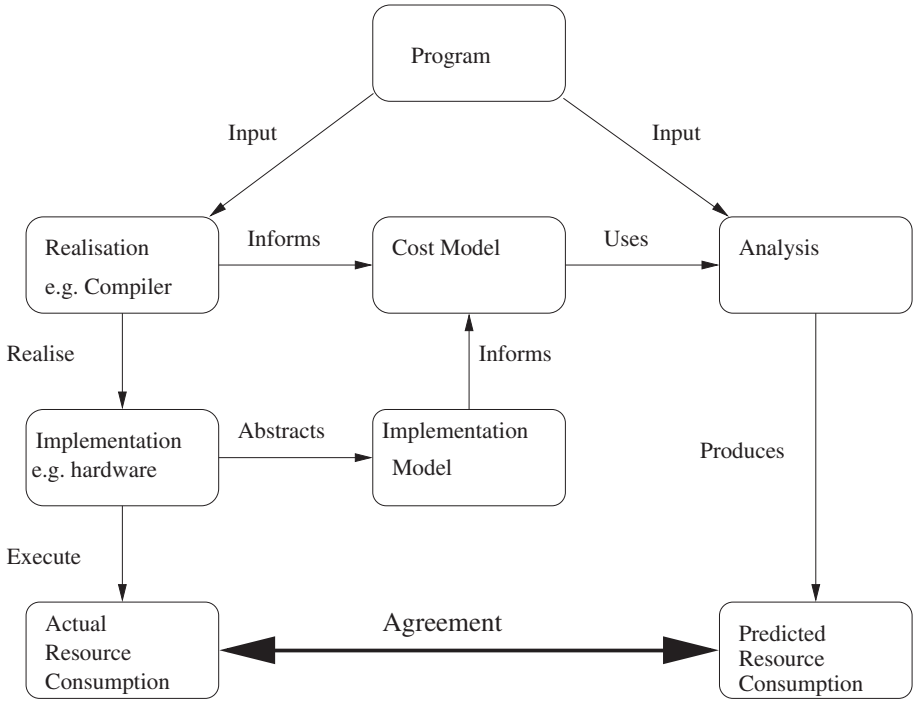

Figure 1. A general model of resource analysis.

3.1. A general model of resource analysis

Figure 1 depicts a general model of resource prediction. The left hand side of the model depicts classical program realisation: a *program* in some language is taken as input by some *realisation*, for example by a compiler or interpreter. The program is realised for some specific *implementation*; for example, a compiler may generate machine code for a specific architecture, or an interpreter may execute on a specific architecture. When the program realisation is executed on that architecture, it will exhibit *resource consumption*. That is, it will consume a certain amount of time, memory, power and other resources, and these can (usually) be measured.

The remainder of the diagram describes the resource analysis, together with its relationship to the program realisation components. The analysis has the following components and information flows.

An *implementation model* specifies the resource consumption of a specific implementation. If time is the resource of interest, the implementation model might record the time to execute each instruction supported by the machine. For space, it might be the space consumed by each instruction. The model may be very accurate (e.g. measuring time as a precise number of machine cycles) or very abstract (e.g. measuring time as a count of function calls).

Cost models. In model theory, a model for a formal language is an *interpretation*, that is assignments to variables that make true some property of some set of sentences. A *cost model* for a programming language has a strongly related but slightly different sense of characterising some cost for an arbitrary program for arbitrary data. A good cost model thus abstracts from the full details of the executable program and operating environment, while capturing enough of their essential characteristics to enable reliable predictions of observable behaviours.

The cost model is parameterised by the implementation model to reflect resource costs on the chosen implementation. It may also be informed by aspects of the language *realisation*, for example information about the compiler optimisations used.

Resource analysis. It is important to distinguish between a cost model, which is an abstract characterisation of program costs or resource usage, and a resource analysis, which is a concrete procedure for determining the cost of a specific program by using some cost model. As discussed in Section 8,

DOI: 10.1002/cpe

an analysis may use one of a variety of techniques, including type inference, abstract interpretation or abstract execution, to build a resource prediction from a cost model.

The output of the analysis is a *predicted resource consumption* that should agree, under some quality conditions, with the actual resource consumption of the program realised on the given implementation. Clearly in a worst-case analysis, it is essential that the actual consumption does not exceed the predicted consumption, but more commonly the similarity between the predicted and actual consumption is measured (e.g. the prediction is within 20% for all programs measured).

3.2. Example sequential resource analysis

We illustrate our general resource analysis model by considering a very simple sequential resource analysis. In Section 4.2, we will show how the model we have built here can be extended to cover the costs of parallel execution.

At its simplest, a model and analysis involve counting how often particular source-level constructs occur in a program execution and then multiplying these counts by some measure. For example, consider the toy expression language $\mathcal{A}$ with the *abstract syntax* shown in Figure 2. The denotational semantics for this language is shown in Figure 3, where *s* maps variables to values, and *value* returns integer values. The *implementation* of this language uses the simple stack machine whose pseudocode instructions are shown in Figure 4. Programs in this language can be *realised* as stack machine instructions by following the compilation rules in Figure 5. For example, we have that

$$\texttt{compa}\ \ [\texttt{a} = \texttt{3}^*(\texttt{b} + \texttt{1})] < \texttt{a} \mapsto 1; \texttt{b} \mapsto 2 > \Rightarrow$$
```
   PUSHI 3; PUSHM 2; PUSHI 1; ADD;
   MULT; POPM 1
```

We may now devise a *cost model* that counts how often each distinct machine instruction is generated, as shown in Figure 6. This cost model is informed by some model of the *implementation*. In this case, $\texttt{INST}_c$ is the abstract cost of machine instruction `INST` on the implementation platform for the resource that we are interested in. In general, the resource might be any monotonically

$$\begin{aligned} &\mathit{assign} \rightarrow \mathit{var} = \mathit{exp} \\ &\mathit{exp} \rightarrow \mathit{var} \mid \mathit{int} \mid \mathit{exp} + \mathit{exp} \mid \mathit{exp} \star \mathit{exp} \mid (\ \mathit{exp}\ ) \end{aligned}$$

Figure 2. A simple arithmetic language, $\mathcal{A}$.

$$\begin{aligned} &\mathrm{s} : \mathit{var} \rightarrow \mathit{int} \\ &\mathrm{ma}\ [\mathit{var}{=}\mathit{exp}]\ \mathrm{s} \rightarrow \quad \mathrm{s}\ (\mathit{var} \mapsto (\mathrm{me}\ [\mathit{exp}]\ \mathrm{s})\ ) \\ &\mathrm{me}\ [\mathit{var}]\ \mathrm{s} \rightarrow \quad \mathrm{s}\ (\mathit{var}) \\ &\mathrm{me}\ [\mathit{int}]\ \mathrm{s} \rightarrow \quad \mathrm{value}(\mathit{int}) \\ &\mathrm{me}\ [\mathit{e1} + \mathit{e2}]\ \mathrm{s} \rightarrow \quad \mathrm{me}\ [\mathit{e1}]\ \mathrm{s} + \mathrm{me}\ [\mathit{e2}]\ \mathrm{s} \\ &\mathrm{me}\ [\mathit{e1} \star \mathit{e2}]\ \mathrm{s} \rightarrow \quad \mathrm{me}\ [\mathit{e1}]\ \mathrm{s} * \mathrm{me}\ [\mathit{e2}]\ \mathrm{s} \\ &\mathrm{me}\ [\ (e)\ ]\ \mathrm{s} \rightarrow \quad \mathrm{me}\ [e]\ \mathrm{s} \end{aligned}$$

Figure 3. Denotational semantics for $\mathcal{A}$.

`PUSHM` *addr*	`stack[sp] = memory[`*addr*`]; sp++;`
`PUSHI` *int*	`stack[sp] = `*int*`; sp++;`
`POPM` *addr*	`memory[`*addr*`] = stack[sp]; sp--; sp--; sp--;`
`ADD`	`stack[sp-2] = stack[sp-2]+stack[sp-1]; sp--;`
`MULT`	`stack[sp-2] = stack[sp-2]*stack[sp-1]; sp--;`

Figure 4. Implementation of $\mathcal{A}$: Stack machine instructions.

DOI: 10.1002/cpe

$s : var \rightarrow addr$

compa [*var* = *exp*] s =	compe[*exp*] s; {POPM s(*var*)}
compe [*var*] s =	{PUSHM s(*var*)}
compe [*int*] s =	{PUSHI *int*}
compe [*e1* + *e2*] s =	compe [*e1*] s; compe [*e2*] s; {ADD}
compe [*e1* ⋆ *e2*] s =	compe [*e1*] s; compe [*e2*] s; {MULT}
compe [(e)] s =	compe [*e*] s

Figure 5. Realisation of $\mathcal{A}$: compilation rules.

costa [*var* = *exp*] =	coste[*exp*] + $\texttt{POPM}_c$
coste [*var*] =	$\texttt{PUSHM}_c$
coste [*int*] =	$\texttt{PUSHI}_c$
coste [*e1* + *e2*] =	coste [*e1*] + coste [*e2*] + $\texttt{ADD}_c$
coste [*e1* ⋆ *e2*] =	coste [*e1*] + coste [*e2*] + $\texttt{MULT}_c$
coste [(e)] =	coste [*e*]

Figure 6. A simple arithmetic language, $\mathcal{A}$.

increasing cost such as time, dynamic memory allocation or power consumption.[‡] Such abstract costs may be considered directly to explore the *relative* resource consumption of program components. Alternatively, given the actual costs for each instruction on some implementation platform (e.g. derived by profiling or from the manufacturer's specifications), we may calculate a predicted cost for a program.

For $\mathcal{A}$, the obvious *analysis* is simple abstract execution of the program by using the cost model, that is an *abstract interpretation* [20] or *symbolic execution*. We note that the analysis is of linear complexity in the number of nodes in a program's abstract syntax tree and that it will always terminate because our source language has no iteration or recursion. The cost for our running example can therefore be calculated as follows:

$$\begin{aligned}&\text{costa}\,[\texttt{a} = \texttt{3}^* \star (\texttt{b} + \texttt{1})] \Rightarrow \\ &\quad 2^*\texttt{PUSHI}_c + \texttt{PUSHM}_c + \texttt{ADD}_c + \texttt{MULT}_c + \texttt{POPM}_c\end{aligned}$$

This section has shown how to construct a simple source-level analysis for determining the resource usage of arithmetic expressions. Our example shows how a language's operational semantics and compiler realisation can guide the construction of a simple *abstract interpretation* of the abstract syntax forms for our expression language. However, this analysis deals only with sequential execution costs.

4. PARALLEL/DISTRIBUTED RESOURCE ANALYSIS

This section introduces parallel/distributed resource analysis, relates it to sequential resource analysis and gives an example parallel analysis. For parallel/distributed resource analysis, coordination costs are a key concern, for example the costs of communicating and synchronising between processes.

We first observe that parallel and distributed resource analyses are instances of the general resource analysis model in Figure 1. For concreteness, let us consider a parallel resource analysis as depicted in Figure 7. The information flows in the parallel model are the same as in the general model; for example, a program is realised on some implementation and consumes resources when executed.

[‡]More complex operations, such as stack usage, require different combining operators to the addition we have used here

DOI: 10.1002/cpe

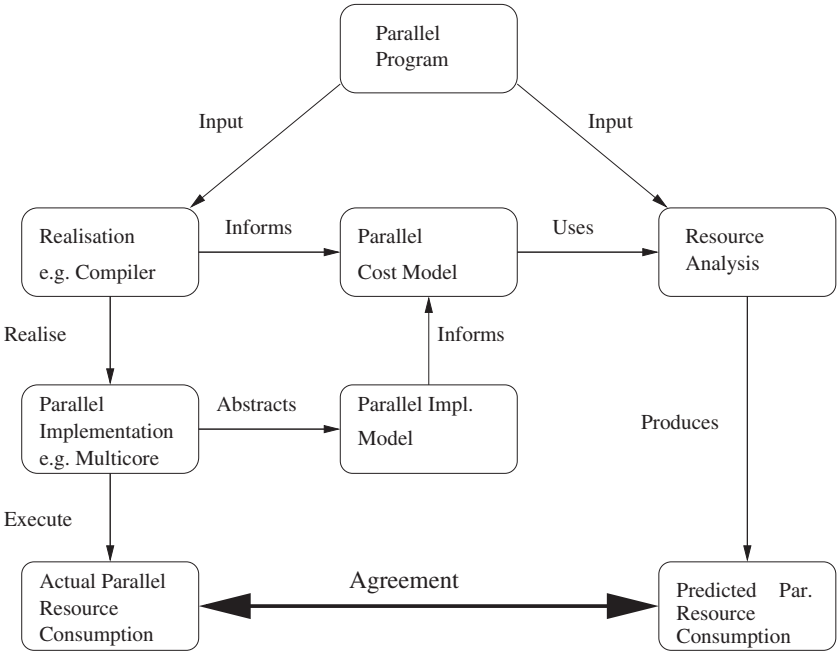

Figure 7. A model of parallel resource analysis.

Crucially however, the realisation, implementation, implementation model and cost model all reflect the parallel coordination aspects. For example, the realisation might compile to multithreaded code, and the implementation might support 128 cores with shared memory. The implementation model might reflect both the number of cores and the costs of creating and synchronising threads on the specific platform.[§] The parallel cost model is parameterised with costs for both sequential program components and for coordination aspects such as communication and synchronisation. In general, these parameters may be obtained by abstraction, profiling, measurement or by resource analysis, as detailed in Section 7.

In contrast to the other components of the model, the analysis component of a parallel analysis typically uses standard techniques to construct the parallel resource consumption prediction. So the analysis might be constructed by abstract interpretation.

4.1. Structuring parallel resource analyses

For the purposes of exposition, let us develop a parallel resource analysis informed by the sequential analysis from the previous section. Although we could simply extend $\mathcal{A}$ and its semantics, with parallel constructs, it is generally better to separate the costs of parallel coordination from those for sequential execution. In such an approach, the parallel/distributed resource analysis is parameterised by information from a sequential resource analysis as shown in Figure 8: we have two instances of the resource analysis model in Figure 1. The parallel implementation comprises a number of sequential components, so the parallel implementation and implementation model are informed by the sequential implementation and implementation model, respectively. Similarly, the parallel cost model coordinates a number of sequential components and hence is informed by the sequential cost model.

The advantages of separating sequential and parallel/distributed analyses is that alternative sequential cost models and analyses can be used, improving the generality of the models and analyses, as well as allowing the reuse of sequential cost models and analyses. It also makes it possible

[§]denoted FORK_c in the example analysis in the following section.

DOI: 10.1002/cpe

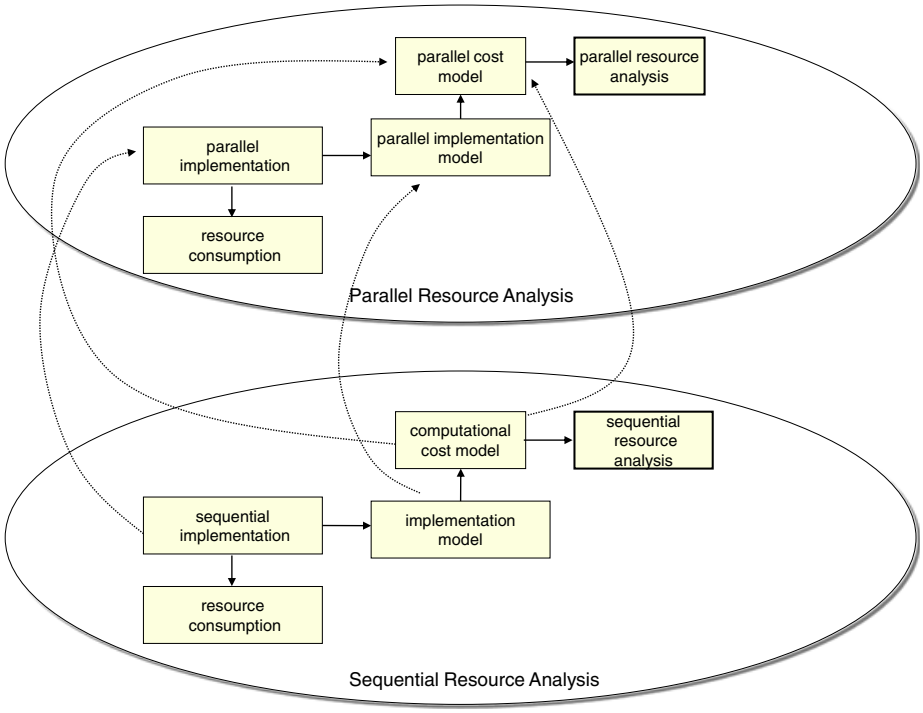

Figure 8. A two-level model: a parallel resource analyses informed by a sequential analysis.

to reason independently about the costs of parallelism, leaving the sequential costs to be supplied through appropriate parameters in the models and analyses.

4.2. Example parallel resource analysis

We define the simple parallel language, $\mathcal{P}$, that is shown in Figure 9. Parallelism is introduced by the `||` construct, where `p||q` executes `p` in parallel with `q`. Similarly, sequential execution is introduced by the `;` construct, where `p;q` executes first `p` and then `q` sequentially. The two independent states that result from executing p and q in parallel are merged using the $\oplus$ operator, whose precise semantics we will leave undefined but which interleave the state changes made by p and q (Figure 10).

Assignments in our sequential arithmetic language $\mathcal{A}$ can be directly embedded in $\mathcal{P}$. We assume that `memory` is shared among all the processors but that each processor will have its own stack pointer and stack, initially empty. We therefore modify each of the stack machine operations to refer to the current processor p as shown in Figure 11. It is also necessary to introduce instructions for parallelism. Here, we use one very simple instruction, `FORK`, which takes two code arguments,

$$\mathit{stmt} \rightarrow \mathit{assign} \quad | \quad \mathit{stmt}\ \texttt{;}\ \mathit{stmt} \quad | \quad \mathit{stmt}\ \texttt{||}\ \mathit{stmt}$$

Figure 9. A simple parallel language, $\mathcal{P}$.

$$\begin{array}{ll} \text{s} : \mathit{var} \rightarrow \mathit{int} & \\ \text{mp}\ [\mathit{st1}\ \texttt{;}\ \mathit{st2}]\ \text{s} \rightarrow & \text{mp}\ [\mathit{st1}]\ (\text{mp}\ [\mathit{st2}]\ \text{s}) \\ \text{mp}\ [\mathit{st1}\ \texttt{||}\ \mathit{st2}]\ \text{s} \rightarrow & \text{mp}\ [\mathit{st1}]\ \text{s} \oplus \text{mp}\ [\mathit{st2}]\ \text{s} \\ \text{mp}\ [\mathit{assign}]\ \text{s} \rightarrow & \text{ma}\ [\mathit{assign}]\ \text{s} \end{array}$$

Figure 10. Denotational semantics for $\mathcal{P}$.

DOI: 10.1002/cpe

```
PUSHM addr | p.stack[p.sp] = memory[addr]; p.sp++;
PUSHI int  | p.stack[p.sp] = int; p.sp++;
POPM addr  | memory[addr] = p.stack[p.sp]; p.sp--;
ADD        | p.stack[p.sp-2] = p.stack[p.sp-2]+p.stack[p.sp-1]; p.sp--;
MULT       | p.stack[p.sp-2] = p.stack[p.sp-2]*p.stack[p.sp-1]; p.sp--;
```

Figure 11. Implementation of $\mathcal{P}$: parallel stack machine instructions.

FORK *task1 task2* | t = *new thread* (*task1*) ; *task2* ; *join* t

Figure 12. Implementation of $\mathcal{P}$: fork instruction.

s : *var* → *addr*

compp [*st1* ; *st2*] s → compp [*st1*] s; compp [*st2*] s
compp [*st1* || *st2*] s → FORK (compp [*st1*] s) (compp [*st2*] s)
compp [*assign*] s → compa [*assign*] s

Figure 13. Realisation: compilation rules for $\mathcal{P}$.

costp [*st1* ; *st2*] → costp [*st1*] + costp [*st2*]
costp [*st1* || *st2*] → *max* (costp [*st1*]) (costp [*st2*]) + FORK_c
costp [*assign*] → costa [*assign*]

Figure 14. Parallel cost model I for $\mathcal{P}$: execution time.

costpp [*st1* ; *st2*] → *max* (costpp [*st1*]) (costpp [*st2*])
costpp [*st1* || *st2*] → costpp [*st1*] + costpp [*st2*]
costpp [*assign*] → 1

Figure 15. Parallel cost model II for $\mathcal{P}$: processors needed.

task1 and *task2*, and executes *task1* in parallel with *task2*. As shown in Figure 12, this is implemented using two primitives: the *new thread* primitive creates a new thread *t* and executes the code for that thread on an available processor; the corresponding *join* primitive waits for a thread *t* to terminate. The corresponding compilation rules are shown in Figure 13.

We can now build the *parallel* cost model shown in Figure 14, which calculates the WCET for a program in $\mathcal{P}$ assuming that there are sufficient processors for all tasks created. As discussed in Section 4.1, the parallel cost model builds essentially on our sequential cost model for $\mathcal{A}$, *costa*. In our fork/join model, and with sufficient processors, the worst-case time for parallel execution is the maximum time for executing either task; the time for executing a sequential statement is just the sum of the times for the two substatements. Note that our implementation model now contains an additional coordination parameter, namely the cost of forking a new task, FORK_c. The value of this parameter might be obtained by measurement, estimation or from a detailed model as discussed in Section 7.

As for the sequential analysis of $\mathcal{A}$, the obvious *analysis* for $\mathcal{P}$ is abstract interpretation. So for example parallel execution time can be predicted as follows.

$$\begin{aligned}&\text{costp}\,[\texttt{a} = 3 * (\texttt{b} + 1) \;||\; \texttt{c} = 5] \Rightarrow \\ &\quad max\,(2{*}\texttt{PUSHI}_c + \texttt{PUSHM}_c + \texttt{ADD}_c + \texttt{MULT}_c + \texttt{POPM}_c, \\ &\qquad \texttt{PUSHI}_c + \texttt{POPM}_c) + \texttt{FORK}_c\end{aligned}$$

It is easy to construct alternative resource analyses. For example, the cost model in Figure 15 calculates the maximum number of processors needed to achieve maximal parallelism in a $\mathcal{P}$ program.

DOI: 10.1002/cpe

4.3. Effectiveness of a parallel/distributed resource analysis

As we have seen, for a resource analysis to be *effective*, it must have some a priori information about the language realisation. It must also have some a priori information about the target implementation platform. The latter may be obtained in a black-box way by measurement, profiling, abstract interpretation, and so on of primitive program constructs on the target platform. That is, constructing a good resource usage model that covers all the issues that may impact performance (whether derived from hardware, software or the operating system) is crucial to obtaining a good and useful resource analysis. Obviously, the analysis as a whole will only be effective if this basic information is good quality.

Let us return to the challenge of constructing an effective *parallel* program, assuming we have an effective sequential resource analysis. We can analyse the components of a program to identify which might be suitable for multiprocessor implementation and apply Amdahl's Law to costs to check whether parallel execution might bring worthwhile benefit. However, this is only half the story. As discussed previously, if we introduce new communication and coordination constructs to enable multiprocessing, then we must also account for their costs.

The remainder of this paper is centrally concerned with different techniques for assessing parallel/distributed communication and coordination costs, under the assumption that adequate models and analyses are available for sequential program costs. As we shall see, such techniques depend strongly on the communication and coordination abstractions used in both programming languages and runtime systems and vary substantially in the degree of detail that they consider and in the precision that they offer.

5. CONSTRAINED PARALLEL/DISTRIBUTED PROGRAMMING PARADIGMS

Many parallel/distributed programming paradigms are constrained, typically to simplify programming. This section briefly outlines why and how parallel/distributed programming paradigms are constrained and surveys some important constrained paradigms that are covered in subsequent sections.

As we have seen in Section 2, parallel/distributed programming can be difficult because in addition to solving the challenges of specifying a correct and efficient algorithm, the programmer must also specify the effective *coordination* of the program components. A number of programming paradigms exist that attempt to ease the challenges of parallel/distributed software engineering by constraining the coordination model but, crucially, *not* the computation model, so the paradigms remain *Turing complete*.

Many of these constrained parallel/distributed programming models simultaneously make the resource analysis of programs more tractable. For example, the programming models may constrain the programmer to use *specific coordination patterns*, such as algorithmic skeletons, or the programmer to use a single *coordination* pattern, such as the bulk synchronous parallel (BSP) model. In both cases, this guides the choice of specific cost models, as we will see in the following section.

5.1. Bulk synchronous parallel

The BSP model constrains the programmer to a single coordination pattern that formulates computations as a series of *supersteps* [21]. A BSP computer comprises a number of parallel processors connected by a communication network. Each processor has a local memory and may follow its own thread of computation. Each superstep in the BSP computation comprises three stages:

- *Independent concurrent computation* on each processor using only local values.
- *Communication* in which each processor exchanges data with every other processor.
- *Barrier synchronisation* where all processes wait until all other processes have finished their communication actions.

We discuss a BSP cost model in Section 6.5.

DOI: 10.1002/cpe

5.2. The Bird–Meertens Formalism

The Bird–Meertens Formalism (BMF) [22] constrains the programmer to use a fixed set of higher-order functions (HOFs). BMF is a calculus for deriving functional programs from specifications, with an emphasis on the use of bulk operations across collective data structures such as arrays, lists and trees. Although independent of any explicit reference to parallelism, many of its operations clearly admit potentially efficient parallel implementation. We discuss BMF cost models in Section 6.6.

5.3. Parallel and distributed skeletons

Skeletons constrain the programmer to use a fixed set of commonly occurring coordination patterns [23]. The patterns are abstracted as library or language constructs that implement the coordination. More precisely the skeletons are HOFs[¶] that the application programmer instantiates with *sequential* code. For example, a parallel `map` function will apply any sequential function to every element of a collection in parallel. The programmer's task is greatly simplified as they do not need to specify the coordination behaviour required.

The skeleton model has been very influential, appearing in parallel standards such MPI and OpenMP [24, 25], and distributed skeletons such as Google's MapReduce [26] are core elements of cloud computing. We discuss cost models for skeleton-based approaches in Section 6.7.

5.4. Workflow languages

Workflow description languages such as Business Process Execution Language [27] or Pegasus (planning for execution in grids) [28] constrain the programmer to use a small set of process structuring constructs such as sequencing and parallelism. Whereas the computations coordinated in the workflow are almost invariably expressed in a Turing complete language, the process structuring constructs provided by most workflow languages are not Turing complete; for example, they may lack iteration.

6. PARALLEL AND DISTRIBUTED COST MODELS

Cost models for parallel and distributed execution range from theoretical models that are aimed at studying algorithmic complexity, such as the parallel random access machine (PRAM) model, down to highly concrete models that may help with static task mapping or dynamic scheduling decisions. In this section, we survey a number of well-known and representative cost models that are used in the analysis methodologies that we will present in Section 8. A more detailed survey of parallel cost models is given in [29].

As shown in Figure 1, cost models for parallel or distributed execution are generally structured as two-level models: the *implementation model* determines execution costs for individual operations; the *high-level model* then synthesises the overall system costs from the implementation level costs, given some parallel execution model for the system as a whole. As we shall see in Section 9, classical sequential cost models can also be useful here, for example in using the predicted execution time for tasks to inform scheduling decisions. Clearly, where a high-level model is built on a low-level one in this way, any inaccuracies in the low-level model will be reflected in, and perhaps magnified by, the high-level model. It is therefore vitally important that the low-level cost model is sufficiently accurate to allow good costs to determined by the high-level model and that the high-level model takes into account any vagaries or deficiencies of the low-level model that could be magnified in the final combined analysis.

[¶]That is, functions that take other functions as arguments, or return functions as their result, effectively comprising templates for generating code.

DOI: 10.1002/cpe

6.1. The parallel random access machine model

The PRAM model [30] is a highly abstract model of parallel computation. It is widely used within the parallel algorithms and complexity research community as a standard theoretical model of a parallel machine, occupying a status similar to that of the Turing machine. By abstracting over primitive PRAM operations, it is possible to derive a standard measure of the parallel complexity of a program. The resulting model is a very general one. In fact, it has even been suggested that PRAM should form a general model of computation, both sequential and parallel [31].

In its simplest form, the PRAM model enforces stepwise synchronous, but otherwise unrestricted, access to a shared memory by a set of conventional sequential processors. At each synchronous step, every processor performs one operation from a simple, conventional instruction set. Each such step is costed at unit time, whatever the operation, and regardless of which shared-memory locations are accessed.

For example, given the problem of summing an array of n integers on a p-processor machine, a simple algorithm A might first sum a distinct subarray of size n/p items on each of the p processors and then use a single processor to sum all the p partial results from the first phase. An informal PRAM cost analysis would capture the cost of this algorithm as follows:

$$T_A(n) = \Theta\left(\frac{n}{p} + p\right)$$

A more sophisticated algorithm B might instead sum the partial results in a parallel tree-like structure (usually known as 'parallel reduction'). An informal PRAM analysis would capture this cost as follows:

$$T_B(n) = \Theta\left(\frac{n}{p} + \log p\right)$$

These analyses clearly show that although algorithm B is asymptotically faster than algorithm A, as intended, this is only true for large p.

Although the basic PRAM model provides a durable and sound basis for at least the initial phases of parallel algorithm design, it ignores a number of important issues such as contention, the memory hierarchy, the underlying communication infrastructure and all internal processor issues. Several variants of the basic PRAM model that aim to address these and other pragmatic cost issues have been introduced. For example, the exclusive-read-exclusive-write PRAM model disallows steps in which any shared-memory location is accessed by more than one processor (algorithms A and B both satisfy this requirement). In contrast, the concurrent-read-concurrent-write PRAM model removes this restriction, with subvariants defining the required behaviour when clashes occur.

6.2. Parallel random access machine models for multicore machines

The PRAM model described previously has been one of the most influential parallel cost models. Because it is highly idealised, for example assuming zero-cost memory access, it is a good basis for a *design-time* analysis (i.e. a manual complexity analysis that is performed by the parallel algorithm designer), where it can be used to expose the maximum parallelism in the algorithm. However, it does not reflect many of the important costs incurred when executing the algorithm on a real parallel machine. Several refinements of this highly abstract model have been suggested, for example the hierarchical PRAM [32], local memory PRAM [33] and block PRAM [34] models.[‖] These variants add the concepts of data locality and remote memory, with varying memory access costs, to the basic PRAM model. Even more detailed are the Parallel Memory Hierarchy (PMH) [35] and Parallel Hierarchical Memory [36] models, which model the entire memory hierarchy. For PMH, the memory hierarchy is modelled as a tree, whose nodes represent memory and whose leaves represent processors. The cost of data transfer in this model is represented as the length of the path

[‖]These are examples of *bridging models*, so called because they comprise a bridge between the high-level programming model and the low-level architecture cost model. See [29] for a general framework for models of parallel computation.

DOI: 10.1002/cpe

between two nodes in the tree. Parameters that characterise the (memory) nodes are the block size, the number of blocks and the transfer time (latency) to neighbouring nodes. This design permits modern, deep memory hierarchies to be accurately modelled.

6.3. Cost models for hierarchical parallel machines

All of these models assume the use of a shared memory, albeit with varying memory access costs. A different class of models has been developed for dealing with distributed memory systems, as found in clusters or clouds. In these models, a conceptual distinction is made between accessing memory locally and transmitting data over a network. The key system characteristics that are commonly modelled in such models include the following:

Degree of parallelism:	the number of processors available.
Latency:	the time between sending a message on one processor and receiving it on another processor. In modern architectures, this usually depends on the depth of the hierarchy.
Memory access:	the cost of reading or writing to a memory location. In modern architectures, this usually depends on the location in the hierarchy.
Synchronisation:	the costs of synchronising a group of processors.
Bandwidth:	the amount of data that can be sent within a given time interval.

One of the best known models is LogP [37], which models the costs of data transfer across a network using the following parameters:

- L ('latency') – the variable amount of time that is needed for communication between two processors.
- o ('overhead') – the fixed amount of time that is needed to prepare for sending or receiving messages.
- g ('gap') – the minimal interval between sending two messages (this is the inverse of the bandwidth).
- P ('processors') – the number of processors on the machine.

The LogP model has proven to be a good compromise between very abstract, simplifying models such as PRAM, and very concrete and detailed models of homogeneous networked architectures. Like the PRAM model, it is mainly used as basis for a design-time analysis. One limitation to the LogP model is that it assumes a homogeneous architecture, where the costs of communication are the same between all processors in a system. Although this may be true for a single cluster, it will generally not be the case for computational grids or cloud computers. An extension of the LogP model, HLogGP [38], has been developed that uses vectors of the LogP parameters to model heterogeneous architectures, such as these. This model has been shown to deliver good predictions on heterogeneous clusters.

We now use the basic LogP model to analyse the costs of our example program of computing the sum over an array of length n on P processors, using list-structured communication. In the LogP model, we have to account for the overhead involved in sending a message and the gap between sending messages. We assume 0 costs for splitting the array into chunks of size $\lceil n/P \rceil$ and a steps as cost for performing an addition. In the broadcast phase of the algorithm, assuming $o < g$, the root processor sends chunks of the array at step $o, o+g, o+2g, \ldots$ in the execution. The last chunk is sent at step $o + g\,(\lceil n/P \rceil - 1)$. All P processors do their summations in parallel. The cost of the summation over one chunk is $c = a\,(\lceil n/P \rceil - 1)$. Computation can begin at step $o + L$ after sending it because L steps are needed for the transmission and o steps for receiving the data. The result of the computation is sent at step o after finishing the computation. Thus, the processors send their results at steps $3o + L + c, 3o + L + g + c, \ldots$ back to the root processor. The last message arrives at $4o+2L+c+g\,(\lceil n/P \rceil - 1)$ steps because it takes L steps to arrive and o steps to process. Computing the overall sum over P partial sums takes aP steps. Thus, the total time for computing the result, based on list-structured communication, is $4o + 2L + c + g\,(\lceil n/P \rceil - 1) + aP$.

As can be seen from this example, the costs incurred by communication and coordination are modelled more accurately compared with the simpler PRAM model. In particular, the explicit

DOI: 10.1002/cpe

parameters for the communication overhead and the gap between sending messages, representing a limit on the bandwidth, give a more realistic picture of the execution and avoid an algorithm design that makes excessive use of interprocessor communication.

6.4. System-oriented cost models

The implementation models that are described in Section 7 can be used to provide detailed information about the costs of coordination operations on a given hardware. Such models are often used in hardware design to capture the characteristics of a parallel machine and to compare idealised performance profiles. They are less attractive as a computational model, however, because they expose many machine-level details to the program and may therefore make it considerably more complicated.

The role of *system-oriented* cost models is to capture the costs imposed by software-level system operations, for example the costs of thread creation. In typical *bridging models*, all such costs are subsumed into a small set of parameters. This simplifies the manual, design-time analysis of algorithms. In contrast, capturing system costs in a separate system-oriented cost model provides additional information that can be used during runtime to control the management of the parallelism. In particular, for systems that perform automatic *load balancing*, where tasks are dynamically moved between processors to equalise the load on each processor, this provides important input to the decision about where a thread should be executed.

In terms of precision, then, system-oriented cost models form an intermediate step between architectural cost models and computational cost models. They are mainly used to construct runtime analyses that can advise a suitably adapted runtime system. For example, they may be used to guide one of the following dynamic resource policies:

Load distribution:	this aims to spread the available parallelism evenly among all processors, with the objective of achieving optimal utilisation of the parallel machine.
Data locality:	this policy aims to keep logically related threads on the same processor to reduce communication costs.
Scheduling:	this policy decides which of the runnable threads to execute next on a given processor.

The following example illustrates how a simple system-oriented cost model can be used to automatically control the load-balancing policy for a heterogeneous architecture. A good strategy for load balancing for tightly coupled multicore processors is to send work to a processor with the highest relative speed $R_i = S_i / W_i$, where S_i is the speed of processor i and W_i is its current load. Thus, work should be transferred from processor j to processor i so that

$$\begin{aligned} &\forall m.\, m \in \{1 \ldots n\}, j \neq m \Rightarrow \\ &\quad (R_m \leqslant R_i) \ \wedge \ \left(k \cdot R_j \leqslant R_i\right) \end{aligned}$$

A throttling factor k is used to avoid overly aggressive work redistribution.

6.5. The bulk synchronous parallel cost model

The BSP model outlined in Section 5.1 occupies a more concrete position in the cost model spectrum than the PRAM models described previously. Like the basic PRAM model, BSP uses a synchronous stepwise model of parallel execution. In contrast to the basic PRAM model, however, the BSP model recognises that synchronisation is not free, that sharing of data involves communication (whether explicitly or implicitly) and that the cost of this communication, both in absolute terms and relative to that of processor-local computation, can be highly machine dependent. It also generalises the sequential computations to be any required computation rather than a small set of primitive operations.

By restricting the programming model to a sequence of supersteps (Section 5.1), it is possible to construct a relatively simple (and accurate) cost model for BSP computations because the cost of a system can be composed from the cost of each superstep.

DOI: 10.1002/cpe

The BSP cost model has two parts: one to estimate the cost of a superstep and another to estimate the cost of the program as the sum of the costs of the supersteps. The cost of a superstep is the cost of the longest running local computation, plus the largest cost of communication between the processes, plus the cost of the barrier synchronisation. The costs are computed in terms of three abstract parameters that respectively model the number of processors p, the cost of machine-wide synchronisation L and g, a measure of the communication network's ability to deliver sets of point-to-point messages, with the latter two normalised to the speed of simple local computations.

For example, the array summing algorithms, translated for BSP, would have asymptotic runtimes of

$$T_A(n) = \Theta\left(\frac{n}{p} + p + pg + 2L\right)$$

where the first two terms are contributed by computation, the third term is contributed by communication and the fourth term is contributed by the need for two supersteps, and

$$T_B(n) = \Theta\left(\frac{n}{p} + \log p + 2g \log p + L \log p\right)$$

where the first term corresponds to local computation, and the other three terms to computation, communication and synchronisation summed across the $\log p$ supersteps of the tree-reduction phase. This analysis clearly reveals the vulnerability of the algorithm B to architectures with expensive synchronisation costs, that is a large L.

This constrained model of parallel computation allows BSP implementations to provide a benchmark suite that derives the implementation model, that is the machine-specific values for the four BSP parameters. These can then be inserted into the abstract (architecture independent) cost derived already for a given program to predict its true performance.

Whereas the BSP model makes no attempt to account for processor internals, the memory hierarchy (other than indirectly through benchmarking) or, for specific communication patterns,[**] a considerable literature testifies to the pragmatic success of the approach [39]. A primary limitation is that because the cost of a superstep is governed by the worst-case cost of any local computation, the BSP model is only suitable for computations where each superstep is fairly regular, that is where the *granularity* of each local computation in a superstep is broadly the same.

6.6. The Bird–Meertens Formalism cost model

A number of authors have investigated adding parallel cost analyses to the BMF-inspired programming models that were outlined in Section 5.2. Two examples are described in the following.

BMF-PRAM: In [40], Cai and Skillicorn present an informal PRAM-based cost model for BMF across list-structured data. Each operation is provided with a cost, parameterised by the costs of the applied operations (e.g. the element-wise cost of an operation to be mapped across all elements of a list) and data structure sizes, and rules are provided for composing costs across sequential and concurrent compositions. The paper concludes with a sample program derivation for the *maximum segment sum* problem. In conventional BMF program-calculation style, an initially 'obviously' correct but inefficient specification is transformed by the programmer into a much more efficient final form. For our array summing problem from Section 6.1, algorithm A would be expressed as a *map* across the partitioned input list, followed by a sequential second phase (perhaps concealed as a further degenerate *map* across a list with only one member, itself the list of partial sums). Analysis would return the result described in Section 6.1, being the sum of the costs of the two phases. The cost of the first phase would emerge from analysis of the mapped function (summing a sublist) and the generic cost of a parallel *map*, being the product of the cost of one instantiation of the mapped function (here n/p) and the number of such calls assigned to each processor (here one).

[**]Indeed, classical BSP implementations rely on randomisation to deliberately *obliterate* patterns in the interests of predictability.

DOI: 10.1002/cpe

Similarly, for algorithm B, the second phase would be expressed as a parallel *reduce* and thereby analysed to cost log p times the cost of the reduction operator (here addition, therefore costing unit PRAM time).

Similarly, in [41], Jay *et al.* build a formal cost calculus for a small BMF-like language using the PRAM model as the underlying cost model. For implementation to be aided, the language is further constrained to be *shapely*, meaning that the size of intermediate bulk data structures can be statically inferred from the size of inputs. The approach is demonstrated by automated application to simple matrix–vector operations. These approaches can be characterised as being of relatively low accuracy (a property inherited from their PRAM foundation), offering a quite rich, although structurally constrained source language, being entirely static in nature and with varying degrees of formality and support.

BMF-BSP: Building on the seminal work described previously, a number of authors have sought to inject more realism into the costing of BMF-inspired parallel programming frameworks by substituting the BSP model for the PRAM model [41, 42]. In particular, [42] defines and implements a BMF-BSP calculus and compares the accuracy of its predictions with the runtime of equivalent (but hand-translated) BSP programs. With the utilisation of maximum segment as a case study, the predictions exhibit good accuracy and would lead to the correct decision at each stage of the program derivation. For our array summing problem, the analysis would return a concrete prediction, composed similarly to the discussion in 6.5 but embedding concrete BSP cost values for the chosen architecture. Meanwhile, in a more informal setting reflecting the approach of [40], Bischof *et al.* [43] report on a BSP-based, extended BMF derivation of a program for the solution of tridiagonal systems of linear equations. Once again, good correlation between (hand-generated) predictions and real implementation is reported, with no more than 12% error across a range of problem sizes. These developments can be characterised as offering enhanced accuracy, while retaining similarly structured models and support. As a by-product of the use of BSP, analyses can now be made specific to the target architecture once they are instantiated with the standard BSP constants for that architecture.

6.7. Skeletons

The *skeleton*-based approach to parallel programming outlined in Section 5.3 advocates that commonly occurring patterns of parallel/distributed computation and interaction should be abstracted as library or language constructs. Several authors have sought to associate cost models to algorithmic skeletons and to use these either explicitly or implicitly to guide the development and implementation of parallel programs. Few authors, if any, have considered the more complex issue of cost models for distributed skeletons.

For example, on the basis of a simple model of message passing costs, Hammond *et al.* [44] use metaprogramming to build cost equations for a variety of skeleton implementations into a skeleton library for the parallel functional language Eden. This approach allows the most appropriate parallel implementation to be chosen at compile-time given instantiation of some target machine-specific parameters. The paper shows how these parameters can be used to discriminate between four possible parallel variants of a farm skeleton for a Mandelbrot visualisation problem.

Meanwhile, Gava [45] describes an attempt to embed the BSP model directly into the functional programming language ML. At the level of parallelism, the programming model is thus constrained to follow the structure of a BSP superstep (a relatively loose skeleton), whereas computation within a superstep is otherwise unconstrained. Analysis is informal, in the conventional BSP style, but the language itself has a robust parallel and distributed implementation. A reported implementation of an n-body solver once again demonstrates close correlation between predicted and actual execution times.

Finally, the approach proposed by Yaikhom *et al.* [46] presents the programmer with imperative skeletons, each with an associated parallel cost model. The models are defined in the performance-enhanced process algebra [47] and are parameterised by a small number of constants that can be derived by running benchmark code fragments. As in [44], models of competing implementation

DOI: 10.1002/cpe

strategies are evaluated, and the best is then selected. In a novel extension, designed to cater for systems in which architectural performance characteristics may vary dynamically, the chosen model is periodically validated against the actual runtime performance. Where a significant discrepancy is found, the computation can be halted, re-evaluated and rescheduled.

Consider again the problem of summing an array of n integers on p processors. The idealised data-parallel skeleton is as follows:

```
void FARM(int p, int  work(int *,int),
          int *input,int n,int *output)
```

where `p` is the number of processors, `work` is a worker function taking an integer array argument to an integer result, `input` is the input array of integers, `n` is the size of the `input` array and `output` is the output array of length `p`. `FARM` sends a chunk of size `n/p` from `input` to each of the `p` processors, which apply `work` to the chunk. Processor `j` returns an `int` to `FARM`, which stores it in `outputs` at position `j`. Given

```
int sum(int *a,int n)
{  int i,s;
   s=0;
   for(i=0;i<n;i++)
     s=s+a[i];
   return s;
}
```

we might call

```
int A[N],O[N];
...
FARM(P,sum,A,N,O);
result = sum(O,P);
```

to sum array `A` of length `N` on `P` processors via array `O`. Suppose that for some distributed memory parallel architectures, where every node has the same characteristics,

$send_{\text{int}}(n)$ is the cost of sending/receiving n `ints` from/to farmer to/from worker;
$process_{\text{sum}}(n)$ is the cost of applying `sum` to n `ints`.

Then the predicted execution time $T_A(n)$ will be

$$\begin{aligned} &\texttt{p} \times send_{\text{int}}\,(\texttt{n/p}) + \\ &process_{\text{sum}}(\texttt{n/p}) + \\ &\texttt{p} \times send_{\text{int}}\,(\texttt{1}) + \\ &process_{\text{sum}}\,(\texttt{p}) \end{aligned}$$

The coordination terms of such cost expressions must be simplified with care. For example, in the equation above, message latency almost certainly means that the cost of sending two messages – $send_{\text{int}}\ (\texttt{n/p}) + send_{\text{int}}\ (\texttt{1})$ – is very different from sending one, slightly larger message – $send_{\text{int}}\ (\texttt{n/p} + \texttt{1})$.

7. IMPLEMENTATION MODEL

The cost models described in the previous section present an abstract, parameterised view of a parallel/distributed computation. They vary in the level of detail that is being provided, as reflected in the number of parameters that are provided by the model. As depicted in Figure 1, to use the model to predict resource costs requires it to be parameterised by an *implementation model* that characterises the target platform.

DOI: 10.1002/cpe

Some implementation parameters of parallel/distributed cost models are readily available as static characteristics of the target architecture. For example, many parallel cost models are parameterised by the number of processors available. Other parameters are harder to determine, for example the communication latency for n units of data.

Some implementation parameters can be determined using techniques that are well established in the sequential resource analysis community, for example execution time, or memory consumption predictions. For completeness, we briefly discuss these techniques in the following.

Other implementation parameters are specific to parallel/distributed cost models, for example communication costs or synchronisation costs, denoted as L and g in the BSP model. The techniques for determining the values of these specialised implementation parameters are, however, broadly similar to determining the parameters for sequential cost models, namely estimation, profiling/measurement or detailed models. This section outlines these approaches and gives references to examples of the techniques in practice in Section 9.

7.1. Estimation

One very simple approach, suitable for abstract cost models, is to abstract over most of the details involved in the parallel execution and to only provide *estimated* values for the parameters. Although not providing realistic costs, such a model can be useful in providing relative information, for example on the degree of communication involved in an execution. This simplified realisation is used in early execution time analyses such as [48, 49].

7.2. Measurement-based approaches

A more scientific approach, and one that more accurately determines the cost model parameters, is to measure the parameter of interest on the target implementation. Commonly, a suite of representative example applications are either *profiled* or *instrumented* to measure the parameter value. For example, Section 9.1 shows how profiling is used to predict parallel execution times, and Section 9.2.3 contains examples showing how instrumented programs are used to determine the computation and communication parameters for a distributed cost model.

For cost model parameters to be determined more easily and more reliably, some libraries provide ready made instrumentation benchmarks to be run on a new architecture, for example [21, 46]. However, both profiling and instrumented programs can suffer from the problems of accuracy and generality that are described in Section 2.

One way of overcoming the accuracy and generality issues is described by Bonenfant *et al.* [50], who have measured costs of bytecode instructions for an abstract-machine implementation. Because well-constructed bytecodes will expose all the interesting cost parameters, and all possible execution paths can be measured for each individual bytecode, this approach allows a relatively small number of measurements to cover the costs of all possible program executions by combining the costs of the individual bytecodes to reflect the execution paths for a particular program. Although issues of cache and pipeline behaviour, and so on can have an impact on the accuracy of the analysis for more complex processor architectures, the approach is promising for abstract-machine implementations, giving a cheap yet general and fairly accurate methodology for determining program execution costs.

7.3. Detailed models

An even more accurate prediction of a parameter value can be obtained by using a detailed model. Detailed models may be constructed for a range of resources, for example memory or time.

For the purposes of exposition, we consider detailed hardware models that provide precise execution time predictions for low-level code. Such a model might provide timing information in terms of machine clock cycles for each machine instruction. Constructing a hardware-level cost model typically involves examining all of the machine operations that are provided by a processor: taking the hardware specification as input, a hardware expert defines bounds on the costs of each instruction.

DOI: 10.1002/cpe

For accuracy, such models need to account for the complete state of the processor, including, for example, its caches and pipelines.

Whereas many uses of resource analysis do not need such a precise model, this level of detail and accuracy is required for industrial-strength WCET analyses. A detailed survey of WCET analyses is given in [51]. Because these analyses are used to guarantee the safety of real-time systems, such as flight controllers or automotive safety systems, WCET analyses must be *safe* in the sense of always producing upper bounds on execution time. A secondary concern is that WCET analyses should be as precise as possible to avoid over-specifying processor requirements, with consequent cost implications. One example of a WCET analysis that combines these features is AbsInt's **aiT** tool that produces worst-case timing information for sequential C code fragments for a number of processor architectures [52]. The use of **aiT** as part of an analysis giving execution time bounds is outlined in Section 9.3.2.

7.4. Determining memory usage and other costs

The aforementioned discussion has focused on execution time. Generally, this is the resource that is of most interest in parallel or distributed programming. It is also one of the most difficult resources to obtain accurate information about because time usage may be nondiscrete and nonmonotonic, depending on detailed contextual information, including the dynamic state of the processor, such as the caches or pipeline states. In the worst case, it may even be nondeterministic! For example, two cores may impact each others execution times by executing threads that share the same cache. The effect depends on the precise timing of the two threads and the precise hardware implementation. In contrast, memory usage tends to be deterministic and may often be monotonic.

Most of the models and techniques described previously can also be used to determine memory or other resource usage costs with small modifications. For example, Hofmann and Jost originally applied an amortised cost analysis to determine linear bounds on heap allocations and deallocation for a first-order functional language, including recursion [10], and this work has subsequently been extended to cover stack usage [53, 54] and WCET costs [12, 54]. This work has been exploited in the EmBounded project funded by the European Union to produce formal cost models and analyses for heap, stack and time usage for the Hume language (http://www.embounded.org). The amortised cost approach that is used in this work aims to even out costs across different operations. This cost can then be used to assign weights in the types of functions, and so on that reflect the costs of different cases in the input and result data structures.[††] Thus, the cost of a function or program can be determined in terms of the sizes of its input and result data structures. A complementary type-based approach can be used to infer the *sizes* of data structures. Chin and Khoo [55] introduced a type inference algorithm that is capable of computing size information from high-level program source. Vasconcelos and Hammond use a similar technique to infer *sized types* for recursive, polymorphic and higher-order programs [56]. Vasconcelos' PhD thesis [9] extends these approaches using abstract interpretation techniques to automatically infer linear approximations of the sizes of recursive data types and the stack and heap costs of recursive functions. By including user-defined sizes, it is possible to infer sizes for algorithms on nonlinear data structures, such as binary trees.

A number of authors have also recently studied analyses for heap usage in an imperative setting. Albert *et al.* [57] present a fully automatic, live heap-space analysis for an object-oriented bytecode language with a scoped-memory manager. This analysis is not restricted to a certain complexity class but unlike the amortised cost approaches, for example, cannot express data dependencies. Braberman *et al.* [58] infer polynomial bounds on the live heap usage for a Java-like language with automatic memory management but do not cover general recursive methods. Finally, Chin *et al.* [59] present a heap and stack analyses for a low-level (assembler) language with explicit (de)allocation, which is also restricted to linear bounds.

[††]This is achieved by assigning different weights to each of the *constructors* of the data structure.

DOI: 10.1002/cpe

8. CONSTRUCTING RESOURCE ANALYSES

A resource analysis uses some cost model to synthesise a prediction for the execution costs of the given program on some implementation platform, as depicted in Figure 1. This section discusses the techniques used to construct resource analyses in each of the three phases of a program's lifetime: design, compilation and execution (runtime).

This section provides only an overview of analysis methods, as they are generic and well established. That is, rather than being specific to resource analysis, methods such as abstract interpretation or constraint system solving are used in many domains for many different purposes. In consequence, they are covered relatively briefly here, with citations to important papers detailing the analysis methods.

8.1. Design-time analysis

Abstract cost models such as those based on the PRAM [30] model, and to some extent those based on the BMF or BSP models, enable the programmer to reason about costs during program design. Such models often require that the program is expressed using a specific structure, for example as a sequence of supersteps for BSP analysis, or using only the BMF operators to express parallelism. Here, the resource analysis is typically not automated, and the relatively simple cost models enable the programmer to perform the resource analysis using pen and paper. A significant advantage of design-time analysis is that guided by the predictions produced by the analysis, the programmer can relatively cheaply transform the program design before committing to a specific implementation.

A commonly used asynchronous, distributed cost model is LogP [37]. It models the costs for data transfer using the following parameters: L, the latency in sending a message between two machines; o, the overhead of composing and sending the message; g, the minimal gap between sending two messages (this corresponds to the inverse of the bandwidth); and P, the number of processors of the machine. This model has proven to be a good compromise between an abstract, user-friendly model of representing a homogeneous, flat parallel hardware and a more accurate machine model that accounts for interprocessor communication costs.

Being used at design time and nonautomated, these models are deliberately simple. However, for precision to be improved and in particular for more complex parallel hardware to be modelled, several extensions to these basic models have been developed. Extensions that include a hierarchical memory structure to the basic machine model include the following: hierarchical PRAM [32], local memory PRAM [33], PMH [35], LogP [37], HLogGP [38] and LogP-PMH [60]. All of these models aim to provide more realistic costs of memory access in a hierarchical memory model and typically view communication as a generalisation of memory access.

Design-time models are also used to improve the coordination of distributed applications. For example, high-level Petri Nets have been used to describe the workflow and resource consumption of complex Grid applications [61].

8.2. Compile-time (static) analyses

Several techniques have been developed that are capable of statically inferring information about runtime resource requirements. The best known of these are a range of compile time analyses exist, primarily type inference, abstract interpretation and constraint system solving. These techniques may be used either individually or in some combination. Nielson *et al.* [62] provide detailed coverage of these techniques.

Type-and-effect systems. Type-and-effect systems are based on the observation that type inference can be separated into two phases: (i) collecting constraints on type/resource variables and (ii) solving these constraints [63]. Several type-and-effect analyses have been developed that extend standard Hindley–Milner type inference to collect constraints on resources, for example [64–67].

Typically, annotations representing the resources are attached to the types in the language. One example is the concept of sized types [68, 69], where for example a numeric annotation to the list type implies an upper bound on the list length. The usual type inference machinery can be used to

DOI: 10.1002/cpe

check the unannotated types and while doing so collect constraints on the annotations. For inferring resource bounds, these constraints are typically (in)equalities over integer or rational numbers. An additional phase then solves the collected constraints to produce resource bounds. This strand of work has been extended to higher-order sized types [70] and combined with a type-and-effect system, performing region analysis, to give resource bounds on dynamic memory consumption [71].

A concrete example of a type-based analysis is the resource analysis for the Hume language presented in Section 9.3 and discussed in detail in [12]. In this case, however, the meaning of the annotations, which are attached to type constructors, is different from those in sized types. They encode an (linear) upper bound on the resource consumption, of an expression, but not a bound on the size of a data structure. The main advantage of this different meaning of the annotations is improved compositionality of the analysis. Furthermore, it avoids the usage of a max-plus-algebra as underlying representation for resource bounds (with the max operator representing the cost of a conditional) and allows the usage of a fairly simple linear program (LP) solver to find a solution for the collected constraints. Finally, attaching all necessary resource information to a type achieves good modularity because the resource analysis only has to examine the type, not the source code, to extract the relevant information from a function possibly defined in a different module.

Abstract interpretation. Abstract interpretation [20] defines an abstract domain of values, which is typically very small and which is often used to provide purely qualitative information. For example, in strictness analysis, the interesting distinction is simply whether or not an expression is strict and therefore safe to be evaluated before passing its result to a function. By using a richer abstract domain, quantitative information, such as resource consumption, can also be modelled. Functions are mapped to abstract versions of those functions that operate over the abstract domain. The analysis then proceeds by executing these abstract functions and in particular finding fixpoints for recursive functions.

One common problem of abstract-interpretation-based analyses is the need of a finite, and in practice small, abstract domain to guarantee fast termination of the fixpoint generation. This often leads to an extremely simple model of resources, which in turn delivers relatively inaccurate information. This problem of domain size is even more pronounced in the presence of HOFs, which give rise to an exponential increase in domain size. One of the main advantages of abstract interpretation is the fact that many practically useful optimisation techniques have been developed for this process. Consequently, well-developed inference engines that can be used for cost analysis exist, for example the COSTA system [72] and the SPEED system [73]. COSTA is generic with respect to the resources that can be analysed, produces high-quality bounds that are not restricted to a particular complexity class and builds on a high-performance inference engine. A combination of this static approach with a runtime, profiling approach is presented in [74]. AbsInt's **aiT** tool [52], which was described previously, uses abstract interpretation over machine-code fragments derived from C program source, synthesising hardware-level information to give guaranteed bounds on WCET. For accurate bounds to be obtained, the analysis uses a detailed implementation model including the cache behaviour and pipeline structure of the processor.

Section 4.2 gives two very simple examples of parallel resource analyses constructed by abstract interpretation, as shown in Figures 14 and 15. In the first, the abstract domain is execution time, and in the second, the abstract domain is number of processors. A more realistic example is the autonomous mobile program (AMP) analysis outlined in Section 9.2.2 and discussed in detail in [75]. This analysis uses abstract interpretation on a cost semantics for a subset of the Jocaml mobile programming language.

Constraint system solving. Constraint-solving approaches are related to the type inference approaches in that they separate the collection of (general) constraints and the solution of these constraints into different phases. However, this process is not necessarily tied to type inference itself. An example of this approach are several variants of control flow analyses [76, 77].

Depending on the nature of the information being extracted by the analysis, different approaches to finding a solution can be used. In simple cases, ad hoc approaches are sufficient. In most cases, however, finding the least fixed point over the set of constraints, as the best solution, is desirable. For

DOI: 10.1002/cpe

this iteration phase to be optimised, techniques from abstract interpretation, in exploring the domain of (abstract) values, can be applied. In fact, most of the tuning of the analysis is happening at this stage. As a consequence, Shiver's control-flow analyses [76] can be seen as abstract interpretations in a wider sense.

8.3. Runtime (dynamic) analysis

An alternative to a static resource analysis is to execute the program with some sample input and to generate profiling information about the execution, including information on resource consumption (performance profiling). This information is precise because it documents observed behaviour, but it very much depends on the particular choice of input values. This resource information can then be directly used in making decisions about the (parallel) execution of the program, in particular to control the granularity of the parallelism [78, 79]. If the entire process of profiling, tuning parameters and executing the target application is automated, the approach is called auto-tuning. This direction of research is attracting considerable attention at the moment [80–83] because it uses powerful machine learning techniques to tackle the complexity of selecting numerous static or dynamic parameters for one particular parallel machine. A comprehensive discussion of such auto-tuning techniques is beyond the scope of this paper.

Purely *pre facto*, dynamic approaches to extracting resource information are prone to bias on the basis of the chosen input data. Hence, they are often combined with other compile-time approaches to provide a better coverage of a programs co-domain. Examples of such hybrid approaches for automated complexity analysis are the ACE [84] and ACAp [85] systems. Both of these systems take strict, functional programs as their input and produce closed-form expressions over the size of the input arguments based on an abstract cost model. The solution to these expressions then gives upper bounds on the execution time. Other dynamic runtime analyses are often used in conjunction with a static resource analysis, for example to approximate the sizes of key data structures [66, 74]. In this context, they form ancillary analyses to the main resource analysis.

9. APPLICATIONS OF PARALLEL/DISTRIBUTED RESOURCE ANALYSIS

This section outlines three practical applications of resource analysis techniques in a parallel or distributed setting. For each application, we present the key elements of the resource analysis model in Figure 7, namely the cost model, the implementation model and the analysis.

We present results showing that each analysis is *effective*, that is that the analysis improves the coordination, typically improving performance. The applications utilise a range of cost models and analyses and use the resource information for a range of coordination purposes including resource-safe execution, optimising parallel execution and enabling mobility.

We also critique each analysis, investigating why the chosen analysis proves effective for each application.

9.1. Informing skeleton selection in PMLS

The UK parallelising ML with skeletons (PMLS) project [86] developed a fully automated parallelising compiler from almost full Standard ML (SML) to C with MPI. The compiler analysed SML programs to identify HOFs that could be used as loci of potential parallelism. These were then realised by instantiating the corresponding parallel algorithmic skeletons.

9.1.1. PMLS cost model. To determine whether the potential parallelism would deliver useful speedup, the compiler used semantics-directed prototyping. The assumption was that there would be a strong and consistent relationship between the language semantics and the implementations.

9.1.2. PMLS analysis. The SML structural operational semantics (SOS) [87] consists of rules of the form *assumptions*/*conclusions* where assumptions and conclusions have the form

$$environment \vdash expression \Rightarrow value$$

DOI: 10.1002/cpe

Here, *environment* is a set of mappings from variables to values, *expression* is some SML syntactic construct and *value* results from evaluating *expression* using the variable bindings in *environment*.

For example, a let expression takes the form

$$\texttt{let}\ dec\ \texttt{in}\ exp\ \texttt{end}$$

where dec introduces new variable/value bindings for use in exp. For example,

$$\texttt{let x = 1 in x + 1 end}$$

binds `x` to `1` and returns `x + 1` that is `2`.

Then the rule for a let expression is

$$\frac{E \vdash dec \Rightarrow E' \qquad E + E' \vdash exp \Rightarrow v}{E \vdash \texttt{let}\ dec\ \texttt{in}\ exp \Rightarrow v\ \texttt{end}}$$

That is, establishing the new variable/value bindings from dec in the environment E gives a new environment E', evaluating the expression exp in the environment E augmented with the new environment E' gives value v and then evaluating the overall let expression in the environment E gives the value v.

For prediction of parallelism to be enabled, a linear model that related composed sequential rule counts to concrete execution times was developed. For some program P_i

$$R_{i,1}W_1 + R_{i,2}W_2 + \ldots R_{i,N}W_N = T_i$$

Here, T_i is the predicted time for P_i on some architecture, $R_{i,j}$ is the number of times SOS rule j fires during sequential semantics-based execution of P_i and W_k is a coefficient relating the contribution that rule k makes to the overall time. Thus, summing products of observed sequential counts and rule weights gives the predicted time.

9.1.3. PMLS implementation model. The ML kit interpreter, which is strongly based on the SML SOS, was modified to collect counts of how often each rule was invoked during sequential execution of a program. Equation coefficients were then found for a specific architecture by instrumenting a large test suite directly on the interpreter and by compiling and timing the test suite on a target architecture node. Then if P is an $r * n$ matrix of observed counts for r programs, x is a vector of corresponding observed target times and w is the vector of unknown coefficients; the linear equation

$$Pw = x$$

was solved for w.

When processing a HOF call in a new program, the function argument could be instrumented on the interpreter and the counts plugged into the solved equations to produce a predicted sequential execution time. This time was then used as an argument to a standard skeleton cost model to predict the parallel execution time. Once all HOFs had been analysed, the compiler would then try to find an optimal skeleton configuration for the program as a whole.

The compiler proved highly portable, and a number of realistic programs were parallelised for the Fujitsu AP3000, Cray T3D, IBM SP2 and Beowulf class architectures.

The main technical challenge was in finding reliable solutions to the linear equations. Two approaches were explored to address this challenge: (i) single-value decomposition and (ii) genetic algorithms. Single-value decomposition found solutions very quickly but was highly sensitive to the test suite. In contrast, the genetic algorithm found highly stable solutions but took an unacceptably long time to converge. Overall, the conclusion was that instrumenting structured operational semantics gave a poor basis for predicting absolute execution times but was well suited for making relative comparisons in choosing between candidate skeleton configurations [88].

DOI: 10.1002/cpe

Table I. Profiler output.

V	Position	HOF	T_p	D_w	D_r	T_m	P_1	P_2	P_4	P_8
1	Outer 1	Map	25.1	81	88	18.7	0.0012	0.0029	0.0061	0.0126
	Inner 2	Map	8.3	243	196	4.1	0.0019	0.0036	0.0068	0.0133
	Inner 3	Fold	9.4	369	351	4.5	0.0106	0.0141	0.0213	0.0355
2	Outer 1	Fold	20.2	128	129	25.5	0.0019	0.0036	0.0068	0.0133
	Inner 2	Map	8.3	243	196	4.1	0.0062	0.0097	0.0169	0.0311
	Inner 3	Fold	9.4	369	351	4.9	0.0106	0.0141	0.0213	0.0355
3	Outer 1	Map	26.0	81	88	19.1	0.0012	0.0029	0.0061	0.0126
	Inner 2	Fold	9.2	287	240	3.9	0.0110	0.0146	0.0217	0.0360
	Inner 3	Fold	9.4	369	351	4.4	0.0106	0.0141	0.0213	0.0355
4	Outer 1	Fold	26.5	128	129	19.4	0.0110	0.0146	0.0217	0.0360
	Inner 2	Fold	9.2	287	240	3.9	0.0062	0.0097	0.0169	0.0311
	Inner 3	Fold	9.4	369	351	4.4	0.0106	0.0141	0.0213	0.0355
5	Outer 1	Map	25.0	81	88	18.7	0.0012	0.0029	0.0061	0.0126
	Inner 3	Fold	9.4	369	351	4.6	0.0106	0.0141	0.0213	0.0355
6	Outer 1	Fold	25.4	128	129	19.2	0.0062	0.0097	0.0169	0.0311
	Inner 3	Fold	9.4	369	351	4.4	0.0106	0.0141	0.0213	0.0355
7	Outer 1	Map	19.3	81	88	16.0	0.0012	0.0029	0.0061	0.0126
	Inner 2	Map	8.3	243	196	3.8	0.0019	0.0036	0.0068	0.0133
8	Outer 1	Fold	19.8	128	129	15.9	0.0019	0.0036	0.0068	0.0133
	Inner 2	Map	8.3	243	196	4.0	0.0062	0.0097	0.0169	0.0311
9	Outer 1	Map	20.2	81	88	15.3	0.0012	0.0029	0.0061	0.0126
	Inner 2	Fold	9.2	287	240	4.0	0.0110	0.0146	0.0217	0.0360
10	Outer 1	Fold	20.7	128	129	15.5	0.0110	0.0146	0.0217	0.0360
	Inner 2	Fold	9.2	287	240	4.0	0.0062	0.0097	0.0169	0.0311
11	Outer 1	Map	19.2	81	88	16.9	0.0012	0.0029	0.0061	0.0126
12	Outer 1	Fold	19.7	128	129	14.7	0.0062	0.0097	0.0169	0.0311

HOF, higher-order function.

Table II. Parallel runtimes for nested implementations.

Version	HOFs	R_2	R_4	R_8
1	MMF	0.595	15.69	7.954
2	FMF	0.886	11.51	5.759
3	MFF	0.600	8.655	4.602
4	FFF	0.882	8.844	5.108
5	MF	0.578	1.724	0.535
6	FF	0.875	2.336	0.651
7	MM	0.267	1.684	0.738
8	FM	0.840	4.449	2.035
9	MF	0.270	1.389	0.699
10	FF	0.598	1.952	0.799
11	M	0.310	0.127	0.090
12	F	0.590	0.167	0.112

HOFs, higher-order functions.

9.1.4. PMLS results. Michaelson and Scaife [89] report results for 12 different realisations of matrix multiplication using various combinations of `map` and `fold`.

Table I shows the predictions for each version squaring a 3×3 matrix. T_p is the sequential instance function execution time averages in μS. D_w is the total number of bytes generated as arguments to the instance function. D_r is the total number of bytes generated as results from instance functions. For comparison, T_m is the measured instance function execution times from the sequential runs.

Table I also shows the predictions (P_2, P_4, P_8) of the runtime in seconds for each HOF in the synthesised programs at two, four and eight processors. These suggest that the most likely candidate is Version 11, a single non-nested `map`.

DOI: 10.1002/cpe

The actual parallel runtimes on two, four and eight processors are shown in Table II, for a 50×50 matrix. For this application, the single non-nested `map` from version 11 gives the best actual performance. These runtimes broadly correspond to the predictions from the profiler.

9.1.5. PMLS resource analysis critique. The distinctive features of the PMLS approach are that

- decisions about exploiting parallelism are governed by cost equations for higher-order constructs instantiated with sequential profile information;
- profiling is abstract and based on semantics;
- a simple resource analysis gives useful information about the *relative* costs of computations; and
- a high degree of cross platform portability is achieved.

9.2. Autonomous mobile programs

Autonomous mobile programs (AMPs) are mobile programs that periodically use a cost model to decide where to execute in a network. AMPs have been constructed and measured in mobile Java variants and in Jocaml, and more complete descriptions of AMP resource analyses and performance can be found in [75, 90].

9.2.1. Autonomous mobile program cost model. The AMP cost model informs the decision whether to move to a new location. The key criterion is whether the predicted time to complete execution on the current location is greater than the predicted time to communicate to the best available location and complete execution there. This is captured in inequality (1) in Figure 16, which shows the relevant elements of the AMP cost model.

9.2.2. Autonomous mobile program analysis. Resource analyses for AMPs have been constructed in two ways. We have constructed an automatic *static* analysis that uses abstract interpretation on a cost semantics for a core subset of Jocaml. The subset includes iterating HOFs such as `map`. Rather than predicting the time to evaluate a term, the model predicts the *continuation* cost of every subterm within a term. This information is used to estimate the time to complete the program from the current point in the execution [75].

$$T_h > T_{comm} + T_n \tag{1}$$

$$T_e = W_d / S_h \tag{2}$$

$$T_h = W_l / S_h \tag{3}$$

$$T_n = W_l / S_n \tag{4}$$

$$W_a = \sum W_d \tag{5}$$

$$W_d = W_{a(thistime)} - W_{a(lasttime)} \tag{6}$$

$$W_l = W_{all} - W_a \tag{7}$$

T_e	:	time that has elapsed at current location
T_h	:	time will take here
T_n	:	time will take in the next location
T_{comm}	:	time for a single communication
W_{all}	:	all work
W_a	:	the work that has been done
W_d	:	the work done at the current location
W_l	:	the work left to do
S_h	:	the current CPU speed
S_n	:	the next location CPU speed

Figure 16. Core distributed cost model for AMPs.

DOI: 10.1002/cpe

```
for i = 0 to n-1 do
  for j = 0 to n-1 do
    for k = 0 to n-1 do
      m3.(i).(j) <- m3.(i).(j) + m1.(i).(k) * m2.(k).(j)
    done
  done
done;;
```

Figure 17. Naive Jocaml matrix multiplication.

$$W_{all} = n^3 \tag{8}$$

$$W_a = f(i,j,k) = (i-1)n^2 + (j-1)n + k \tag{9}$$

$$W_l = W_{all} - W_a = n^3 - f(i,j,k) = n^3 - (i-1)n^2 + (j-1)n + k \tag{10}$$

$$W_d = W_{a(thistime)} - W_{a(lasttime)} = f(i,j,k)_{thistime} - f(i,j,k)_{lasttime} \tag{11}$$

$$T_e = \frac{W_d}{S_h} = \frac{[f(i,j,k)_{thistime} - f(i,j,k)_{lasttime}]\, Sec}{S_h} \tag{12}$$

$$T_h = \frac{W_l}{S_h} = \frac{\left[n^3 - f(i,j,k)\right] Sec}{S_h} \tag{13}$$

$$T_h = \frac{\left[n^3 - f(i,j,k)\right] T_e}{f(i,j,k)_{thistime} - f(i,j,k)_{lasttime}} \tag{14}$$

$$T_n = \frac{W_l}{S_n} = \frac{\left[n^3 - f(i,j,k)\right] Sec}{S_n} = \frac{S_h T_h}{S_n} \tag{15}$$

Figure 18. AMP matrix multiplication cost model.

A far simpler *design-time* analysis is to parameterise the AMP cost model with a sequential cost model for the AMP algorithm. This is an instance of the two-level approach described in Figure 8 where the sequential analysis is a simple analytical model.

We illustrate the design-time analysis using the naive Jocaml matrix multiplication shown in Figure 17. The sequential analysis is readily constructed; for example, the total work, W_{all}, is n^3, and the work carried out so far is a function of i, j and k, that is $(i-1)n^2 + (j-1)n + k$. The full matrix multiplication cost model is shown in Figure 18. In Equations (12) and (13), Sec is a constant that converts abstract units of work to elapsed time (seconds) on a given architecture.

9.2.3. Autonomous mobile program implementation model. We use profiling to determine the cost model parameters for the target architectures. Predicting computation time entails determining the Sec constant for the target processor. This can be carried out by solving Equation (13) given the sequential execution time of the program T_h for a matrix size n. We can validate the computation component of the model as follows. At every iteration of the top level matrix multiplication loop, we use Equation (14) to predict the remaining time and the total time for the program. At the end of the program, we can get the real execution time and compare the predicted time and the real time. Table III shows that we achieve accurate predictions of computation times.

To determine the communication costs, we assume that communication time is a function of the size of the matrix(n^2). So we suppose the time for communication has the form

$$T_{comm} = T_{setup} + T_{unit} * n^2 \tag{16}$$

Here, T_{setup} is the set-up time, that is the time to establish a connection, and T_{unit} is the time to send a unit of data. From experiments, we find that the set-up time is constant. The communication

DOI: 10.1002/cpe

Table III. AMP execution time validation.

Size	Pred (s)	Act (s)	Standard deviation (%)
$500 * 500$	5.63	5.72	1.7
$600 * 600$	9.75	9.86	1.2
$700 * 700$	15.75	15.57	1.2
$800 * 800$	23.00	23.30	1.3
$900 * 900$	32.40	32.97	1.7
$1000 * 1000$	45.25	45.72	1.0

Table IV. AMP communication time validation.

Data Size	Pred (s)	Act (s)	Standard deviation (%)
$20 * 20$	0.029	0.028	5.0
$50 * 50$	0.042	0.047	12.5
$100 * 100$	0.081	0.079	2.3
$200 * 200$	0.236	0.259	9.9
$300 * 300$	0.495	0.510	3.0
$400 * 400$	0.857	0.840	2.0
$500 * 500$	1.323	1.276	3.5

time only grows if the matrix is larger than 50. For $n > 50$, a constant is obtained if the communication time is divided by n^2. So for a 100 Mb/s ethernet, we get the following equation, and Table IV shows its validation.

$$T_{comm} = 0.029 + \left(\text{if n } < 50 \text{ then } 0 \text{ else } 5.07 * 10^{-6} * n^2\right).$$

9.2.4. Autonomous mobile program results. The AMP resource analysis is used dynamically; that is, Equation (1) is periodically evaluated using the present speed of the current location S_h and the speed of the fastest available alternative S_n.

Autonomous mobile programs can exploit the cost model to move to a faster location and hence reduce execution time. Figure 19 shows the execution times of matrix multiplication programs. Our test environment is based on three locations with CPU speed 534 MHZ, 933 MHZ and 1894 MHZ. The loads on each locations is almost zero. We start both the static and mobile programs on the first, slowest location.

Although AMPs act independently to minimise their execution time, collections of AMPs perform dynamic load management without a load manager. Figure 20 shows the distribution of 10 Java Voyager AMPs across four identical locations (y-axis) over time (x-axis). Initially, seven AMPs are started at a single location, and they quickly spread evenly over the locations in the network. The

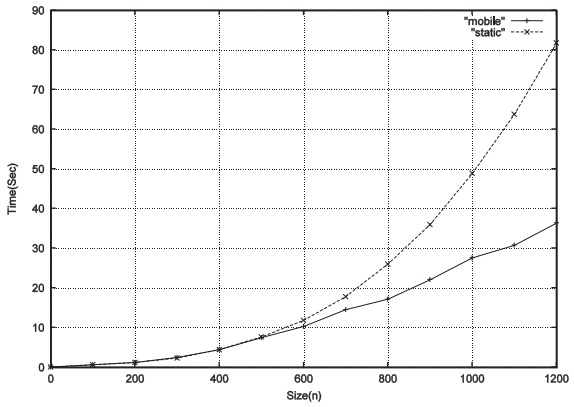

Figure 19. Mobile and static matrix multiplication execution time.

DOI: 10.1002/cpe

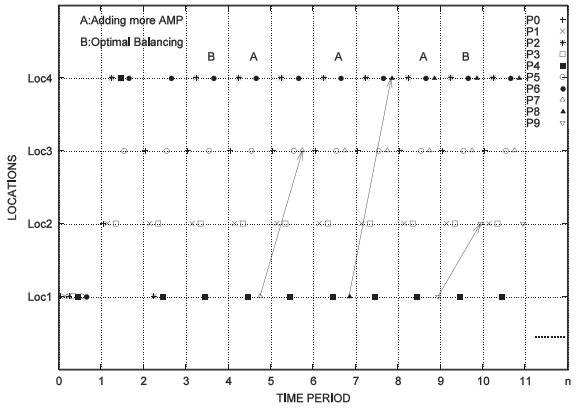

Figure 20. Managing 10 AMPs on four homogeneous locations.

Java Voyager implementation places additional overheads on the initiating location; hence, it retains just one AMP, whereas the other locations have two AMPs each by time period 3. In subsequent time periods, three AMPs are started at the initiating location, and they distribute themselves over the network to reach a new balanced state in time period 9.

9.2.5. Autonomous mobile program resource analysis critique. The rather simple abstract costed operational semantics used by AMPs is effective for a combination of reasons.

- It compares the *relative* cost of completing at the current location with the cost of completing at an alternative location.
- It requires only *coarse grain execution time estimates*. That is, rather than attempting to predict the execution time of small computational units, it compares the time to complete the entire program on the current location with the time to complete on an alternative location.
- It *incorporates dynamic information* into the static model, that is parameterising the model with current performance.

9.3. Resource-safe execution in Hume

The goal of resource-safe execution is to statically guarantee that available resources are never exhausted. In this setting, the role of a resource analysis is to provide concrete *upper bounds* on resource consumption rather than just predictions. Furthermore, formal guarantees of the validity for these bounds enhance the confidence in the results and open the possibility of automatic checks.

Resource bounds have a wide range of applications. A classical sequential example is to ensure that a long running program on an embedded device will never exhaust available resources. Resource bounds are also valuable in many parallel/distributed contexts, for example to inform task placement across a range of heterogeneous architectures from multicores to Grids. Another example is to attach resource bounds to mobile code to certify the resource implications of executing the code.

9.3.1. Hume cost model. The abstract cost model, underlying the Hume resource analysis, is formulated as a resource-annotated operational semantics, where costs are attached to all Hume instructions. This choice was made to facilitate the formal soundness proof of the resource analysis with respect to the underlying semantics. The relevant information for resource consumption, as encoded in the semantics, is extracted into a table of basic costs, each of which corresponds (a part of) a Hume instruction. This table is then a parameter of the generic resource analysis.

The following judgement of the operational semantics

$$\Sigma; \mathcal{V}, \eta \vdash^{t}_{t'} e \rightsquigarrow \ell, \eta'$$

reads as follows: The expression e evaluates under the configuration $\Sigma; \mathcal{V}, \eta$ in a finite number of steps to a result value stored at location ℓ in heap η', provided that there were t time units available

DOI: 10.1002/cpe

before computation. Furthermore, at least t' time units are unused after the evaluation is finished. Again, we will omit to name Σ explicitly for the sake of brevity because the signature is fixed for any given program and because we assume a fixed but arbitrary program.

As an example, we give the operational semantics for the conditional construct in Hume in the following. Note that constants `Tiftrue` and `Tiffalse` are used to abstractly model the costs of conditional branching. Different cost tables are used to model heap space, stack space or execution time, without changing the structure of the rules. In total, our cost model uses 46 parameters, resulting in a fine-grained and thus very accurate abstract cost model.

$$\frac{\eta\big(\mathcal{V}(x)\big) = (\mathsf{bool}, \mathtt{tt}) \qquad \mathcal{V}, \eta \overset{t - \texttt{Tiftrue}}{\underset{t'_1}{\vdash}} e_1 \rightsquigarrow \ell', \eta'}{\mathcal{V}, \eta \overset{t}{\underset{t'_1}{\vdash}} \texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rightsquigarrow \ell', \eta'} \text{ (CONDITIONAL TRUE)}$$

$$\frac{\eta\big(\mathcal{V}(x)\big) = (\mathsf{bool}, \mathtt{ff}) \qquad \mathcal{V}, \eta \overset{t - \texttt{Tiffalse}}{\underset{t'_2}{\vdash}} e_2 \rightsquigarrow \ell', \eta'}{\mathcal{V}, \eta \overset{t}{\underset{t'_2 - \texttt{Tgoto}}{\vdash}} \texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \rightsquigarrow \ell', \eta'} \text{ (CONDITIONAL FALSE)}$$

The modelling of resources in this operational semantics is unusual and deserves further explanation. Conceptually, the semantics maintains a counter, which starts with the total number of available resources and counts *down* in the rules of the semantics. The costs in the true case are $t - Tiftrue - t1'$, that is the difference in the counters before and after executing the conditional. The resource variables, attached to the `then` branch indicate that must be a bound on its consumption. Hence, the total consumption is at least `Tiftrue` plus the costs for the `then` branch. Note that this formula is not made explicit but rather encoded as a side condition in the semantics. This is crucial for the machinery of the analysis, which collects inequality constraints and then uses an efficient LP solver (lp_solve [91]) to find a solution. We will see in the example of running the analysis that only the overall result is then elaborated in a more user-friendly way. Correspondingly, the annotations in the false case can be interpreted as a bound of `Tiffalse` + `Tgoto` plus the costs for the `else` branch.

9.3.2. Hume implementation model. The implementation model [92] is an *accurate hardware model* of the time (in cycles) and space (in bytes) consumption of compiled Hume code. By applying a separate, machine-code-level WCET analysis on each operation of the underlying abstract machine, the cost parameters of the operational semantics are instantiated with provable upper bounds. Thus, it can be characterised as a *safe cost model*. This results in a *concrete cost model*, of about 30 entries. Because the low-level analysis takes into account hardware characteristics such as cache and pipeline structure, the resulting bounds are very precise.

The following paragraphs elaborate on how the parameters for the cost model, in particular those for time, have been established and assessed with respect to concrete measurements. This concrete instance of a cost model is for a Renesas M32C/85 microprocessor.

Structure of the low-level analysis. The **aiT** tool is a machine-level, abstract-interpretation-based static analysis that determines the WCET of a program task in the following phases: building a control-flow graph from an executable binary program, performing a value analysis on instructions accessing memory, performing a cache analysis to predict cache usage, performing a pipeline analysis to predict processor behaviour, and finally performing a path and WCET analysis.

The cache analysis phase uses the results of the value analysis phase to predict the behaviour of the (data) cache on the basis of the range of values that can occur in the program. The results of the cache analysis are then used within the pipeline analysis to allow prediction of those pipeline stalls that may be due to cache misses. The combined results of the cache and pipeline analyses are used to compute the execution times of specific program paths. By separating the WCET determination into several phases, it becomes possible to use different analysis methods that are tailored to the specific subtasks. Value analysis, cache analysis and pipeline analysis are all implemented using abstract interpretation [20]. Integer linear programming is then used for the final path analysis phase.

DOI: 10.1002/cpe

Although the analysis works at a level that is more abstract than simple basic blocks, it is not capable of managing the complex high-level constructs of the Hume language. It can, however, provide useful and accurate worst-case time information about lower level constructs. This motivates our design of using the accurate, low-level **aiT** analysis to determine bounds on the machine code, corresponding to Hume's abstract machine, and to then use these values as constants in a cost table by our type-based, high-level resource analysis for Hume.

Quality of the static analysis using the aiT tool. To assess the quality of the low-level analysis bounds, we have compared average execution and WCETs, for Hume abstract-machine instructions compiled using a commercial C compiler (IAR). Each average and worst-case entry has been obtained from 10 000 individual timings. The worst-case times and average-case times are very similar for most instructions, indicating that the instruction timings are highly consistent in practice. Because certain instructions are parameterised on some argument, in these cases we have measured several points and applied linear interpolation to obtain a cost formula. It is interesting to note that in these case, the linear factor is identical for both WCET and average times, and the constants are also very close.

For the instructions we have compared, the bound given by the static analysis is at most 50% greater than the measured worst case; the mean difference is 22%, with a standard deviation of 16%. We conclude that the static analysis provides an accurate upper bound on execution time.

9.3.3. Hume analysis. The resource analysis for Hume [12, 54], together with the infrastructure for executing Hume code on embedded systems, is an instance of such resource-safe execution. The source language, Hume, has two layers. The box layer defines a network of boxes that communicate along single-buffer one-to-one wires. The expression layer is a *strict, higher-order functional language*. The resource analysis is a *static, type-based analysis*, building on the concept of amortised costs [93]. It produces, where possible, linear bounds on the resource consumption. The restriction of this techniques to linear bounds has in the meantime been relaxed to polynomial bounds [94]. Some supported resources are heap-space and stack-space consumptions and worst-case execution time.

The central notion in amortised cost analysis is that of a '*potential*', that is the amount of resources that is (conceptually) available to the program to spend. Each operation reduces this potential. This is formally expressed by attaching the potential available before and after the execution to the type of a program expression (variables $\mathtt{t}$ and $\mathtt{t}'$ in the rule that follows). Inferring the overall resource consumption therefore amounts to performing type inference over this extended type system, finding a solution for the variables encoding the potential at each point in the execution of the program. Informally, the overall resource consumption is the difference between the potential available at the beginning and those at the end of the execution. In his PhD thesis, Jost [95] develops the foundations for this analysis, including a formal soundness proof, discusses implementation aspects and assesses the quality of the resource bounds.

The type rules for expressions have the form

$$\Sigma; \Gamma \vdash^{m}_{m'} e : A \mid \psi$$

where Γ is a context mapping program variables to enriched Hume types, m, m' are resource expressions, e is the Hume expression, A is an enriched Hume type and ψ is a set of constraints involving resource variables and constant rational values. More precisely, the meaning of this statement is as follows: for all valuations ν mapping all resource variables to rational values such that the constraint set $\nu(\psi)$ is satisfied, the Hume expression e has type $\nu(A)$ in the context $\nu(\Gamma)$. Note that ν applied to an annotated type (or type environment) instantiates the resource variables with concrete, rational values, leaving the structure of the type unchanged. Furthermore, for all memory configurations consisting of environment $\mathcal{V}$ (mapping variables to addresses) and heap η (mapping addresses to values, fitting the (type) context Γ), executing the expression e will require at most $\nu(m) + \Phi_\eta(\mathcal{V} : \nu(\Gamma))$ time units. A time unit is usually defined as a single clock cycle of the processor, but other units such as nanoseconds are also possible if desired. Intuitively, the function $\Phi_\eta(\mathcal{V} : \nu(\Gamma))$ over a heap η, an

DOI: 10.1002/cpe

environment $\mathcal{V}$ and a type context Γ calculates the overall potential encoded by the free variables in expression e. This overall potential is the sum of the weights of all constructors that are reachable from these free variables, where the weight of each constructor in the sum is determined by the type of the variables in the type environment Γ, instantiated by the valuation ν.

Furthermore, if the computation finishes with heap η' and result value v, then at least $\nu(m') + \Phi_{\eta'}(v : \nu(\Gamma))$ time units remain unused after the computation has finished. This notion of unused time units is required for compositionality.

As an example, we now give the rule for the conditional in Hume.

$$\frac{\Gamma \vdash^{\frac{t - \texttt{Tiftrue}}{t'}} e_t : A \mid \phi \qquad \Gamma \vdash^{\frac{t - \texttt{Tiffalse}}{t' + \texttt{Tgoto}}} e_f : A \mid \psi}{\Gamma, x{:}\texttt{bool} \vdash^{\frac{t}{t'}} \texttt{if}\ x\ \texttt{then}\ e_t\ \texttt{else}\ e_f : A \mid \phi \cup \psi} \quad (\textsc{Conditional})$$

The rules of the extended type system define the type of the expression, in this case A, and collect a set of equality constraints over rational variables, in this case $\phi \cup \psi$. The potentials before and after the conditional are represented by the variables t and t'. The constant `Tiftrue` represents the cost of checking the head of the conditional, which must be a variable in this intermediate format of the Hume language, in the true case. Thus, this cost is deducted from the potential before entering the `then` branch, and the potentials before and after the `then` branch are $t - \texttt{Tiftrue}, t'$ (similarly, for the `else` branch). The constraints from both branches, represented by ϕ and ψ, are combined on top level. There is no simplification of the constraint set in this phase because solving the constraints with a state-of-the-art LP solver is very fast for realisitic examples.

9.3.4. Hume results. In this section, we present several examples of our analysis of simple, expression-level Hume programs.

Example: sum-over-list. The first example infers the costs for a list-traversing function, computing the sum over a list of float values.

```
type _float = float 32;

data flist = Cons _float flist | Nil;

sum11 :: flist -> _float;
sum11 (Nil) = 0.0 ;
sum11 (Cons f fs) = f + (sum11 fs);

expression sum11;
```

The intermediate code for this example shows how a function with pattern matching is translated into (possibly nested) case statements. The overloaded multiplication operation on Hume level is instantiated to a monomorphic $*.$ over floats.

```
program

type flist = Cons {-2-} float  flist | Nil {-0-}

-- type of main
val main :: flist, ->float

-- Functions
{sum11 :: flist, ->float (?arg_11 :: flist) =
 case ?arg_11 of
   ((Nil)) -> 0.0|
   (Cons f  fs) -> glet ?z_1    = (sum11 <> fs)
                   in f+.?z_1
```


DOI: 10.1002/cpe

```
 esac}
-- Boxes
-- Expression:
sum11
```

Unsurprisingly, the time consumption of the main expression, namely the function `sum11`, is linear in the length of the input list, as shown by the following (rich) type:

```
ARTHUR3 typing for resource "Time":
  30, (flist[934;float<0>,#|0]) -476/126-> float<0> ,0
```

For every `Cons` node of the list, represented as `float<0>,#` in the type, 934 cycles are needed to perform the computation. In total, this gives a worst-case time consumption of $934n + 476$ for an input list of length n.

Red–black trees. As a second example, we analyse heap consumption inserting a node into a red–black tree, shown in Figure 21. This example is directly taken from Okasaki's textbook [96] and discussed in more detail in [97]. A red–black tree is a binary search tree, in which nodes are coloured red or black. With the help of these colours, invariants can be formulated that guarantee that a tree is roughly balanced. The invariants are that on each path no red node is followed by another red node and that the number of black nodes is the same on all paths. These invariants guarantee that the lengths of any two paths in the tree differs by at most a factor of two. This loose balancing constraint has the benefit that all balancing operations in the tree can be performed locally. The `balance` function only has to look at the local pattern and restructure the tree if a red–red violation is found. The `rbInsert` function in Figure 21 performs the usual binary tree search, finally inserting the node as a red node in the tree (if it does not already exist in the tree) and balancing all trees in the path down to the inserted node.
The heap bound for the `rbInsert` function, inferred by our analysis is

```
ARTHUR3 typing for HumeHeapBoxed:
 (int,tree[Leaf|Node<10>:colour[Red|Black<18>],#,int,#]) -(20/0)->
           tree[Leaf|Node:colour[Red|Black],#,int,#]
```

```
type num = int 16;
data colour = Red | Black;
data tree = Leaf | Node colour tree num tree;

balance :: colour -> tree -> num -> tree -> tree;
balance Black (Node Red (Node Red a x b) y c) z d =
   Node Red (Node Black a x b) y (Node Black c z d);
balance Black (Node Red a x (Node Red b y c)) z d =
   Node Red (Node Black a x b) y (Node Black c z d);
balance Black a x (Node Red (Node Red b y c) z d) =
   Node Red (Node Black a x b) y (Node Black c z d);
balance Black a x (Node Red b y (Node Red c z d)) =
   Node Red (Node Black a x b) y (Node Black c z d);
balance c a x b = Node c a x b;

ins :: num -> tree -> tree;
ins x Leaf = Node Red Leaf x Leaf;
ins x (Node col a y b) = if (x<y)      then balance col (ins x a) y b
                         else if (x>y) then balance col a y (ins x b)
                         else               (Node col a y b);

rbInsert :: num -> tree -> tree;
rbInsert x t = case ins x t of (Node _ a y b) -> Node Black a y b;
```

Figure 21. Example `rbInsert`: insertion into a red–black tree.

DOI: 10.1002/cpe

Using the option `--speak` to elaborate the resource bound, encoded in the aforementioned type information, we obtain

```
ARTHUR3 typing for HumeHeapBoxed:
(int,tree[Leaf<20>|Node<18>:colour[Red|Black<10>],#,int,#])
 -(0/0)-> tree[Leaf|Node:colour[Red|Black],#,int,#]

Worst-case Heap-units required to call rbInsert in relation to
its input:
  20*X1 + 18*X2 + 10*X3
    where X1 = number of "Leaf" nodes at 1. position
          X2 = number of "Node" nodes at 1. position
          X3 = number of "Black" nodes at 1. position
```

This bound expresses that the total heap consumption of the function is $10n + 18b + 20$, where the n is the number of nodes in the tree, l is the number of leaves and b is the number of black nodes in the tree. The latter demonstrates how our analysis is able to produce data-dependent bounds by attaching annotations to constructors of the input structure. This gives a more precise formula compared with one that only refers to the size of the input structure. In this example the $18b$ part of the formula reflects the costs of applying the `balance` function, which restructures a subtree with a black root in the case of a red–red violation. The analysis assumes a worst case, where every black node is affected by a balancing operation. Note that because of the aforementioned invariants, this cannot occur for a well-formed red–black tree: any insertion into the tree will trigger at most two balancing operations. As expected, these (semantic) constraints are not captured by our analysis: our analysis must account for the worst case of all well-typed programs. However, the *type* of red–black trees does not capture such semantic conditions and includes malformed trees (e.g. a tree with all nodes being red is still well typed), whose processing must thus be accounted for.

Resource analysis critique. The following characteristics of the resource analysis for Hume [54] make it an effective tool for delivering guaranteed resource information.

- The analysis is purely static, and thus, resource-safe execution can be guaranteed before executing the code.
- For such guarantees to be delivered, the type-based analysis builds on strong formal foundations, and the type system is proven sound [12].
- Through its tight integration of resource information into the type system, by using numeric annotations to types, it is natural to base the static analysis on a type inference engine.
- To guarantee that the analysis delivers bounds, we must start with a precise and safe cost model, itself representing upper bounds.
- For tight upper bounds to be facilitated, the analysis uses an accurate hardware model.

The key requirement in this application domain is safety, and thus, the emphasis is on the formal aspects of the analysis. Beyond these aspects, the following practical aspects contribute to the usability of the inferred resource information.

- Through the generic treatment of resources, the analysis can be easily retargeted for other (quantitative) resources.
- By using a standard LP solver in the constraint-solving stage, we achieve a relatively fast analysis.

10. CONCLUSIONS

This paper has discussed the value of resource predictions for parallel and distributed systems (Section 2), presented a general model of resource analysis (Section 3), described parallel/distributed resource analysis and its relationship to sequential resource analysis (Section 4), surveyed resource analysis for parallel and distributed computing (Sections 5–8) and provided a critical evaluation of three representative parallel/distributed resource analyses (Section 9).

DOI: 10.1002/cpe

10.1. Characterising effective resource analyses

On the basis of our experience of parallel/distributed resource analysis, including applications in Section 9, we can identify the following general principles that govern why the chosen combination of techniques is effective in each case. The principles represent general guidance rather than a recipe for how to design an analysis for a specific application.

- In general, the analyses are tailored to the requirements of the specific application area. This involves considering a number of issues, including the following:
  - The analysis must deliver information that is sufficiently accurate. Sometimes, a surprisingly simple cost model can be effective, for example for AMPs (Section 9.2).
  - The analysis must usually combine both static and dynamic components in a suitable way. For some applications, purely static information suffices, where others require at least some dynamic information. For example, AMPs dynamically parameterise a static cost model with current network loads (Section 9.2.4).
  - In many cases, it is sufficient for the analysis to produce qualitative predictions, for example whether it is worth creating a thread to evaluate an expression. However, in some scenarios, such as resource-safe execution, the analysis must produce guaranteed bounds on resource usage (Section 9.3).
- Highly abstract resource analyses, such as BMF-PRAM, can be informative even at early phases of parallel algorithm design (Section 6.6).
- More refined, architecture-dependent analyses can be utilised either during the design phase or during subsequent stages of the parallel program development process (Section 6.6).
- Improving the analysis for reusable coordination abstractions, such as algorithmic skeletons or parallel libraries, can have a significant impact on the applications that use them (Section 9.1).
- Even partial cost information can prove useful. For example, in a parallel analysis, we may generate parallelism for computations with predicted execution time above some threshold and ignore any computations for which we are unable to produce a prediction.
- Frequently, quantitative resource predictions are used in a qualitative way, and hence, imprecise or relative resource information is sufficient. Examples include where quantitative predictions are used to chose between alternative PMLS parallelisations (Section 9.1.4) and for AMPS to chose between execution on alternative locations (Section 9.2.1).
- For resource analysis to be facilitated, modern language design often integrates cost modelling as an essential part of the design process, for example in NESL [98] and Hume [99] (Section 9.3).

10.2. Future trends in resource analysis

Some important trends that we anticipate in the near future are as follows. There are trends towards type-based analysis and also towards the enrichment of standard type systems with information on resource consumption [94, 100]. The machinery that is already used for basic type inference in a number of programming languages has proved to be very flexible and entirely capable of inferring various kinds of resource usage information in addition to its main use for obtaining type information.

A very active area of research is the auto-tuning of parallel programs, based on profiling information and using machine learning techniques to improve parallel performance [101]. With future parallel architectures likely to be hierarchical and heterogeneous, analytic models of performance prediction will be challenged to deliver accurate predictions of performance. By using observable behaviour, on an input set providing sufficiently wide coverage, choosing static parameters and dynamic control policies based on this behaviour, and iterating this process, learning from the previously observed behaviour is an attractive alternative.

At the same time, automated theorem proving techniques are becoming increasingly mature. A number of type-based approaches have been proposed that expose type information in the form of provable theorems, and these can easily be extended to consider resources in addition to types. This combination enables the generation of formal certificates of bounded resource usage [102–104].

DOI: 10.1002/cpe

In early work on automated complexity analysis, much attention was paid to the step of solving recurrence equations as one key part of an automated analysis. Here, the trend goes away from the usage general-purpose symbolic computation systems to the usage of more specialised solvers that deliver bounds rather than exact solutions and are tuned for patterns typically occurring in automated program analyses [105]. Another approach, motivated by the area of implicit complexity analysis, is to construct an analysis that avoids an explicit solution of recurrence equations but limits the expressable bounds to certain classes of expression, such as linear [12] or polynomial [94] functions over the input sizes.

10.3. Resource analyses and parallel/distributed systems

Parallel and distributed systems are becoming increasingly prevalent, and good resource analysis is becoming increasingly important if parallel/distributed applications are to be developed rapidly and deployed effectively. We anticipate that future parallel and distributed systems will be able to better exploit the rapid improvements in the basic resource analysis technologies that are being made and in the quality of information that they will produce. Indeed, recent advances have already both substantially widened the range of programs for which static information can be provided. They have also made obtaining resource information easier, for example as in the COSTA system [72, 106] that provides an integrated environment combining cost analysis and compilation. We expect this trend to continue. In the future, we will view the use of resource analysis for parallel and distributed systems in much the same way as we view the use of type systems for computations today, that is as an essential and necessary part of our software development process.

ACKNOWLEDGEMENTS

We acknowledge the contributions of Xiao Yan Deng to the AMP resource analysis reported in Section 9.2, Steffen Jost to the Hume resource analysis reported in Section 9.3 and Norman Scaife to the PMLS resource analysis reported in Section 9.1. We also acknowledge the anonymous referees whose comments have significantly improved the paper.

This work has been supported by the European Union grants RII3-CT-2005-026133 ‘SCIEnce: Symbolic Computing Infrastructure in Europe’, IST-2010-248828 ‘ADVANCE: Asynchronous and Dynamic Virtualisation through performance ANalysis to support Concurrency Engineering’, IST-2011-287510 ‘RELEASE: A High-Level Paradigm for Reliable Large-scale Server Software’ and IST-2011-288570 ‘ParaPhrase: Parallel Patterns for Adaptive Heterogeneous Multicore Systems’, and by the UK’s Engineering and Physical Sciences Research Council grants EP/G055181/1 ‘HPC-GAP: High Performance Computational Algebra and Discrete Mathematics’ and EP/F 030657/1 ‘Islay: Adaptive Hardware Systems with Novel Algorithmic Design and Guaranteed Resource Bounds’.

REFERENCES

1. Cohen J, Zuckerman C. Two languages for estimating program efficiency. *Communications of the ACM* 1974; **17**(6):301–308.
2. Wegbreit B. Mechanical program analysis. *Communications of the ACM* 1975; **18**(9):528–539.
3. Ramshaw LH. Formalizing the analysis of algorithms. *PhD thesis*, Stanford University Department of Computer Secience, 1979.
4. Wegbreit B. Verifying program performance. *Journal of the ACM* 1976; **23**(4):691–699.
5. Hickey T, Cohen J. Automating program analysis. *Journal of the ACM* 1988; **35**(1):185–220.
6. Zimmermann W. Automatische Komplexitätsanalyse von funktionalen Programmen (Automatic complexity analysis of functional programs) (in German). *PhD thesis*, University of Karlsruhe, 1990.
7. Flajolet P, Salvy B, Zimmermann P. Automatic average-case analysis of algorithms. *Theoretical Computer Science* 1991; **79**:37–109.
8. Rebón Portillo Á, Hammond K, Loidl H-W, Vasconcelos PB. Cost analysis using automatic size and time inference. In *Proceedings of IFL 2002: Implementation of Functional Languages, Madrid, Spain*, Springer LNCS 2670, 2003.
9. Vasconcelos PB. Cost inference and analysis for recursive functional programs. *PhD Thesis*, University of St Andrews, February 2008.
10. Hofmann M, Jost S. Static prediction of heap space usage for first-order functional programs. In *POPL’03 – Symposium on Principles of Programming Languages*. ACM Press: New Orleans, LA, USA, 2003.
11. Hofmann M, Jost S. Type-based amortised heap-space analysis. *ESOP’06: Proceedings of the 2006 European Symposium on Programming*, Vienna, Austria, 2006; 22–37.

DOI: 10.1002/cpe

12. Jost S, Hammond K, Loidl H-W, Hofmann M. Static determination of quantitative resource usage for higher-order programs. *POPL '10 Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Madrid, Spain, 2010; 223–236.
13. Talpin J-P, Jouvelot P. Polymorphic type, region and effect inference. *Journal of Functional Programming* July 1992; **2**(3):245–271.
14. Amtoft T, Nielson F, Nielson H. *Type and effect systems: behaviours for concurrency*. Imperial College Press: London, UK, 1999.
15. Souyris J. Industrial experience of abstract interpretation-based static analyzers. In *Building the Information Society*, Vol. 156, Jacquart R (ed.), IFIP International Federation for Information Processing. Springer: Boston, 2004; 393–400.
16. Gulwani S, Mehra KK, Chilimbi T. SPEED: precise and efficient static estimation of program computational complexity. In *POPL '09: Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM: New York, NY, USA, 2009; 127–139.
17. Trinder PW, Cole MI, Loidl H-W, Michaelson GJ. Characterising effective resource analyses for parallel and distributed coordination. In *Proc. FOPARA '09: Intl. Workshop on Foundational and Practical Aspects of Resource Analysis*, Springer LNCS 6324. Springer Verlag: Berlin, Germany, 2009; 67–83.
18. Amdahl GM. Validity of the single processor approach to achieving large scale computing capabilities. *AFIPS '67 (Spring): Proceedings of the April 18–20, 1967, Spring Joint Computer Conference*, New York, NY, USA, 1967; 483–485.
19. Foster I. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison Wesley: Boston, USA, 1995.
20. Cousot P, Cousot R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *POPL'77: Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, Los Angeles, California, 1977; 238–252.
21. Hill JMD, McColl DC, *et al*. BSPlib: the BSP programming library. *Parallel Computing* 1998; **24**(14):1947–1980.
22. Bird R, de Moor O. *Algebra of Programming*. Prentice Hall: London, UK, 1997.
23. Cole M. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press: Cambridge, USA, 1989.
24. MPI Forum. MPI 2: extensions to the message-passing interface. *Technical Report*, University of Tennessee, Knoxville, 1997.
25. Chandra R, Menon R, Dagum L, Kohr D, Maydan D, McDonald J. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers: San Mateo, CA, USA, 2000.
26. Dean J, Ghemawat S. *Beautiful Code*, chapter Distributed Programming with MapReduce. O'Reilly: Sebastopol, CA, USA, 2007. ISBN 0596510047.
27. Jordan D, Evdemon J, *et al*. Web Services Business Process Execution Language Version 2.0, 2007. Available from: http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf [August 2011].
28. Lee K, Paton NW, Sakellariou R, Deelman E, Fernandes AAA, Mehta G. Adaptive workflow processing and execution in Pegasus. *Concurrency and Computation: Practice and Experience* 2009; **21**(16):1965-1981.
29. Maggs BM, Matheson LR, Tarjan RE. Models of parallel computation: a survey and synthesis. In *HICSS'95: Proceedings of the 28th Hawaii International Conference on System Sciences*. IEEE Computer Society: Washington, DC, USA, 1995; 61–70.
30. Fortune S, Wyllie J. Parallelism in random access machines. In *STOC '78: Proceedings of the 10th Annual ACM Symposium on Theory of Computing*. ACM: San Diego, California, USA, 1978; 114–118.
31. Vishkin U. A case for the PRAM as a standard programmer's model. In *Parallel Architectures and Their Efficient Use*, Meyer F, Monien B, Rosenberg A (eds). Springer Verlag: Berlin, Germany, 1993; 11–19.
32. Heywood TH, Ranka S. A practical hierarchical model of parallel computation I: the model. *Journal of Parallel and Distributed Computing* 1992; **16**:212–231.
33. Aggarwal A, Chandra AK, Snir M. Communication complexity of PRAMs. *Journal Theoretical Computer Science* 1990; **71**(1):3–28.
34. Aggarwal A, Chandra AK, Snir M. On communication latency in PRAM computations. *Symposium on Parallel Algorithms and Architectures*, Santa Fe, New Mexico, USA, 1989; 11–21.
35. Alpern B, Carter L, Ferrante J. Modeling parallel computers as memory hierarchies. In *Conference on Programming Models for Massively Parallel Computers*. IEEE Computer Society Press: Washington, DC, USA, 1993; 116–123.
36. Vitter JS, Shriver EAM. Algorithms for parallel memory II: hierarchical multilevel memories. *Algorithmica* 1994; **12**:148–169.
37. Culler DE, Karp R, *et al*. LogP: towards a realistic model of parallel computation. *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, (PPoPP93)*, San Diego, CA; 1993.
38. Bosque JL, Pastor L. A parallel computational model for heterogeneous clusters. *IEEE Transactions on Parallel and Distributed Systems* 2006; **17**(12):1390–1400.
39. Bisseling R. *Parallel Scientific Computation: A Structured Approach using BSP and MPI*. Oxford University Press: Oxford, UK, 2004.
40. Skillicorn DB, Cai W. A cost calculus for parallel functional programming. *Journal of Parallel and Distributed Computing* 1995; **28**(1):65–83.

DOI: 10.1002/cpe

41. Jay CB, Cole MI, Sekanina M, Steckler P. A monadic calculus for parallel costing of a functional language of arrays. *Euro-Par '97: Proceedings of the Third International Euro-Par Conference on Parallel Processing, Springer LNCS 1300*, Passau, Germany, 1997; 650–661.
42. Hayashi Y, Cole M. Automated cost analysis of a parallel maximum segment sum program derivation. *Parallel Processing Letters* 2002; **12**(1):95–111.
43. Bischof H, Gorlatch S, Kitzelmann E. Cost optimality and predictability of parallel programming with skeletons. *Proceedings of Euro-Par '03: European Conference on Parallel Processing, Springer LNCS 2790*, 2003; 682–693.
44. Hammond K, Loogen R, Berhold J. Automatic skeletons in Template Haskell. *HLPP '03: Proceedings of the 2003 Workshop on High Level Parallel Programming*, Paris, France, 2003.
45. Gava F. BSP functional programming: examples of a cost based methodology. *ICCS '08: Proceedings of the 8th International Conference on Computational Science, Part I, Springer LNCS*, Kraków, Poland, 2008; 375–385.
46. Yaikhom G, Cole M, Gilmore S. Combining measurement and stochastic modelling to enhance scheduling decisions for a parallel mean value analysis algorithm. In *Proc. Intl. Conference on Computational Science (2)*, 2006; 929–936.
47. Hillston J. *A Compositional Approach to Performance Modelling*. Cambridge University Press: New York, NY, USA, 1996.
48. Bjerner B, Holmström S. A compositional approach to time analysis of first order lazy functional programs. In *FPCA'89 – Conference on Functional Programming Languages and Computer Architecture*. ACM Press: San Diego, California, USA, 1989; 157–165.
49. Rosendahl M. Automatic complexity analysis. In *Proc. FPCA'89: Intl. Conference on Functional Programming Languages and Computer Architecture*. ACM Press: San Diego, California, USA, 1989; 144–156.
50. Bonenfant A, Ferdinand C, Hammond K, Heckmann R. Worst-case execution times for a purely functional language. *IFL'06 Proceedings of the 18th International Conference on Implementation and Application of Functional Languages, Spinger LNCS 4449*, Budapest, Hungary, September 4–6 2006; 235–252.
51. Wilhelm R, Engblom J, Ermedahl A, Holsti N, Thesing S, Whalley D, Bernat G, Ferdinand C, Heckmann R, Mueller F, Puaut I, Puschner P, Staschulat J, Stenström P. The worst-case execution-time problem – overview of methods and survey of tools. *ACM TECS: Transactions on Embedded Computer Systems* 2008; **7**(3):1–53.
52. Ferdinand C, Heckmann R, *et al*. Reliable and precise WCET determination for a real-life processor. *EMSOFT '01 Proceedings of the First International Workshop on Embedded Software, Springer LNCS 2211*, Tahoe City, USA, October 8–10, 2001; 469–485.
53. Campbell B. Amortised memory analysis using the depth of data structures. In *Proceedings of ESOP 2009: European Symposium on Programming, LNCS 5502*. Springer: York, UK, 2009; 190–204.
54. Jost S, Loidl H-W, Hammond K, Scaife N, Hofmann M. "Carbon Credits" for resource-bounded computations using amortised analysis. *FM '09 Proceedings of the 2nd World Congress on Formal Methods, Springer LNCS 5850*, Eindhoven, the Netherlands; 2009.
55. Chin W-N, Khoo S-C. Calculating sized types. *Higher-Order and Symbolic Computing* 2001; **14**(2,3):261–300.
56. Vasconcelos PB, Hammond K. Inferring cost equations for recursive, polymorphic and higher-order functional programs. In *Proc. IFL 2003: Intl. Workshop on Impl. of Functional Languages*, LNCS 3145. Springer: Edinburgh, UK, Sept. 2003; 86–101.
57. Albert E, Genaim S, Gómez-Zamalloa M. Live heap space analysis for languages with garbage collection. *ISMM '09: Proceedings of the 2009 international symposium on Memory management*, Dublin, Ireland, June 19–20, 2009.
58. Braberman V, Fernández F, Garbervetsky D, Yovine S. Parametric prediction of heap memory requirements. *ISMM '08: Proceedings of the 7th International Symposium on Memory Management*, New York, NY, USA, 2008.
59. Chin W-N, Nguyen HH, Popeea C, Qin S. Analysing memory resource bounds for low-level programs. *ISMM '08 Proceedings of the 7th International Symposium on Memory Management*, New York, NY, USA; 2008.
60. Li Z, Mills PH, Reif JH. Models and resource metrics for parallel and distributed computation. In *HICSS'95: Proceedings of the 28th Hawaii International Conference on System Sciences*. IEEE Computer Society: Washington, DC, USA, 1995; 133–143.
61. Alt M, Dumitrescu C, Gorlatch S, Kertesz A, Sipos G, Epema DHJ. Towards user-transparent performance prediction for workflows of higher-order components. *Proceedings of the CoreGRID Integration Workshop*, CYFRONET Poland, 2006; 345–356.
62. Nielson F, Nielson HR, Hankin C. *Principles of Program Analysis*. Springer: Berlin, Germany, 2005.
63. Kuo T-M, Mishra P. Strictness analysis: a new perspective based on type inference. In *Proc. FPCA '89: Intl. Conference on Functional Programming Languages and Computer Architecture*. ACM Press: Imperial College, London, UK, September 11–13, 1989; 260–272.
64. Dornic V, Jouvelot P, Gifford DK. Polymorphic time systems for estimating program complexity. *ACM Letters on Programming Languages and Systems (LOPLAS)* 1992; **1**(1):33–45.
65. Reistad B, Gifford DK. Static dependent costs for estimating execution time. In *Proc. LFP '94: 1994 ACM Conference on LISP and Functional Programming*. ACM: New York, NY, USA, 1994; 65–78.
66. Huelsbergen L, Larus JR, Aiken A. Using the run-time sizes of data structures to guide parallel-thread creation. In *LFP'94: Proceedings of the 1994 ACM Conference on LISP and Functional Programming*. ACM: New York, NY, USA, 1994; 79–90.
67. Loidl H-W. Granularity in large-scale parallel functional programming. *PhD Thesis*, Department of Computing Science, University of Glasgow, March 1998.

DOI: 10.1002/cpe

68. Hughes RJM, Pareto L, Sabry A. Proving the correctness of reactive systems using sized types. In *POPL'96: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM: St. Petersburg Beach, Florida, 1996.
69. Hughes RJM, Pareto L. Recursion and dynamic data structures in bounded space: towards embedded ML programming. In *ICFP'99: Proceedings of the 4th ACM SIGPLAN International Conference on Functional Programming*. ACM: Paris, France, 1999; 70–81.
70. Abel A. A polymorphic lambda-calculus with sized higher-order types. *PhD thesis*, Ludwig-Maximilians-Universität München, 2006.
71. Montenegro M, Pena R, Segur C. A space consumption analysis by abstract interpretation. In *Proc. FOPARA '09: Intl. Workshop on Foundational and Practical Aspects of Resource Analysis*, Springer LNCS 6324. Springer Verlag: Berlin, Germany, 2009; 34–50.
72. Albert E, Arenas P, Genaim S, Puebla G, Zanardini D. COSTA: design and implementation of a cost and termination analyzer for Java bytecode. In *Proceedings of the International Symposium on Formal Methods for Components and Objects (FMCO 2007), LNCS 5382*. Springer: Amsterdam, The Netherlands, October 24–26, 2007; 113–132.
73. Gulwani S, Mehra KK, Chilimbi TM. SPEED: precise and efficient static estimation of program computational complexity. In *POPL '09: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM: Savannah, USA, 2009; 127–139.
74. Mera E, López-García P, Puebla G, Carro M, Hermenegildo M. Combining static analysis and profiling for estimating execution times. In *Proc PADL '07: Intl. Symp. on Practical Aspects of Declarative Languages*, LNCS 4354. Springer: Berlin, Germany, January 2007; 140–154.
75. Deng XY, Trinder PW, Michaelson GJ. Cost-driven autonomous mobility. *Computer Languages, Systems and Structures* 2010; **36**(1):34–51.
76. Shivers O. Control-flow analysis of higher-order languages. *PhD Thesis*, School of Computer Science, Carnegie Mellon University, May 1991.
77. Midtgaard J. Control-flow analysis of functional programs. *ACM Computing Surveys* 2012.
78. Sargeant J. Improving compilation of implicit parallel programs by using runtime information. In *Workshop on the Compilation of Symbolic Languages for Parallel Computers*. Argonne National Laboratory, July 1993; 129–148.
79. Sodan AC, Bock H. Extracting characteristics from functional programs for mapping to massively parallel machines. *HPFC'95 – High Performance Functional Computing*, Denver, Colorado, April 10–12, 1995; 134–148.
80. Tournavitis G, Wang Z, Franke B, O'Boyle MFP. Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09. ACM: New York, NY, USA, 2009; 177–187.
81. Li J, Ma X, Singh K, Schulz M, de Supinski BR, McKee SA. Machine learning based online performance prediction for runtime parallelization and task scheduling. *International Symposium on Performance Analysis of Systems and Software (ISPASS 2009)*, Boston, MA, April 2009; 89–100.
82. Lee BC, Brooks DM, de Supinski BR, Schulz M, Singh K, McKee SA. Methods of inference and learning for performance modeling of parallel applications. In *PPOPP*, Yelick KA, Mellor-Crummey JM (eds). ACM: San Diego, California, USA, 2007; 249–258.
83. Ipek E, de Supinski BR, Schulz M, McKee SA. An approach to performance prediction for parallel applications. In *Euro-Par*, Vol. 3648, Cunha JC, Medeiros PD (eds), Lecture Notes in Computer Science. Springer: Berlin, Germany, 2005; 196–205.
84. Le Métayer D. Mechanical analysis of program complexity. In *SIGPLAN '85 Symposium*, Vol. 20(7), SIGPLAN Notices. ACM Press: New York, 1985; 69–73.
85. Benzinger R. Automated complexity analysis of Nuprl extracted programs. *Journal of Functional Programming* 2001; **11**(1):3–31.
86. Scaife N, Horiguchi S, Michaelson G, Bristow P. A parallel SML compiler based on algorithmic skeletons. *Journal of Functional Programming* July 2005; **15**(4).
87. Milner R, Tofte M, harper R, MacQueen D. *The Definition of Standard ML (Revised)*. MIT: Cambridge, 1997.
88. Scaife N, Michaelson G, Horiguchi S. Empirical parallel performance prediction from semantics-based profiling. *Scaleable Computing: Practice and Experience* September 2006; **7**(3):1–8.
89. Michaelson G, Scaife N. *Functional Prototyping for Parallel Skeleton based Implementation*. Springer-Verlag: Berlin, Germany, 2002. 129–153.
90. Deng XY, Trinder PW, Michaelson GJ. Autonomous mobile programs. *Proceedings of IAT '06: IEEE/WIC/ACM Intelligent Agent Technology*, Hong Kong, 2006; 177–186.
91. Berkelaar M, Eikland K, Notebaert P. lp_solve: Open source (mixed-integer) linear programming system. GNU LGPL (Lesser General Public Licence). Available from: http://lpsolve.sourceforge.net/5.5 [June 2011].
92. Jost S, Loidl H-W, Hammond K. Report on WCET analysis. EmBounded Project Deliverable, February 2007. Deliverable D14.
93. Tarjan RE. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods* April 1985; **6**(2):306–318.
94. Hoffmann J, Hofmann M. Amortized resource analysis with polynomial potential – a static inference of polynomial bounds for functional programs. In *Proceedings of the 19th European Symposium on Programming (ESOP'10)*, Vol. 6012, Lecture Notes in Computer Science. Springer: Berlin, Germany, 2010; 287–306.

DOI: 10.1002/cpe

95. Jost S. Automated amortised analysis. *PhD Thesis*, Faculty of Mathematics, Computer Science and Statistics, LMU Munich, Germany, 2010.
96. Okasaki C. *Purely Functional Data Structures*. Cambridge University Press: Cambridge, UK, 1998. ISBN 0521663504.
97. Loidl H-W, Jost S. Improvements to a resource analysis for Hume. In *FOPARA '09: Intl. Workshop on Foundational and Practical Aspects of Resource Analysis*, LNCS 6324. Springer: Eindhoven, the Netherlands, November 2009.
98. Blelloch GE. Programming parallel algorithms. *Communications of the ACM* 1996; **39**(3):85–97.
99. Hammond K, Michaelson GJ. Hume: a domain-specific language for real-time embedded systems. *GPCE '03 Proceedings of the 2nd International Conference on Generative Programming and Component Engineering , Springer LNCS 2830*, Erfurt, Germany, 2003; 37–56.
100. Danielsson NA. Lightweight semiformal time complexity analysis for purely functional data structures. *POPL '08: Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Francisco, USA, 2008; 133–144.
101. Tournavitis G, Wang Z, Franke B, O'Boyle MFP. Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. *SIGPLAN Not.* June 2009; **44**:177–187.
102. Necula G. Proof-carrying-code. In *Proc. POPL '97: ACM Symposium on Principles of Programming Languages*. ACM: San Diego, CA, USA, January 15–17 1997; 106–116.
103. Crary K, Weirich S. Resource bound certification. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM: Boston, USA, 2000; 184–198.
104. Beringer L, Hofmann M, Momigliano A, Shkaravska O. Automatic certification of heap consumption. *Logic for Programming, Artificial Intelligence, and Reasoning, 11th International Conference, (LPAR04)*, Montevideo, Uruguay, 2004; 347–362.
105. Albert E, Arenas P, Genaim S, Puebla G. Automatic inference of upper bounds for recurrence relations in cost analysis. *Proceedings of the International Symposium on Static Analysis (SAS 2008), Springer LNCS 5079*, Valencia, Spain, July 15–17, 2008; 221–237.
106. Navas J, Mera E, López-García P, Hermenegildo M. User-definable resource bounds analysis for logic programs. In *Logic Programming*, Dahl V, Niemelä I (eds), LNCS 4670. Springer: Berlin, Germany, 2010; 348–363.

DOI: 10.1002/cpe