Characterising Effective Resource Analyses for Parallel and Distributed Coordination

P. W. Trinder[1], M. I. Cole[2], H-W. Loidl[1], and G. J. Michaelson[1]

[1] School of Mathematical and Computer Sciences,
Heriot-Watt University, Riccarton, Edinburgh EH14 4AS, U.K.
{trinder,hwloidl,greg}@macs.hw.ac.uk
[2] School of Informatics
The University of Edinburgh, 10 Crichton Street Edinburgh, EH8 9AB
mic@inf.ed.ac.uk

Abstract. An important application of resource analysis is to improve the performance of parallel and distributed programs. In this context key resources are time, space and communication. Given the spectrum of cost models and associated analysis techniques available, what combination should be selected for a specific parallel or distributed context?
We address the question as follows. We outline a continuum of coordination cost models and a range of analysis techniques. We consider six representative parallel/distributed applications of resource analysis techniques, and aim to extract general principles governing why the combination of techniques is effective in its context.

1 Introduction

Parallel and distributed programs must specify both the computation to be performed, and how this is to be *coordinated* across multiple locations. Effective resource analyses enable better coordination, for example scheduling can be improved with accurate estimates of the computational cost for units of work. The resource analyses need to build on realistic cost models to reflect the resource consumption incurred during execution. Furthermore, an appropriate analysis technique must be used to predict resource consumption to the required accuracy. Finally, there are many possible uses of such resource information.

Section 2 classifies coordination cost models, focusing on the level of abstraction over the hardware that is provided. The PRAM model is extremely simple and abstract. More refined models use a fixed structure of the execution of the code to produce an accurate cost model: Bulk Synchronous Processes (BSP) is one such example. Finally, there is a rich class of models that take hardware details such as caches and processor pipelines into account to produce a very accurate model: for example the processor model used by AbsInt's aiT analysis for (sequential) worst case execution time.

Section 3 classifies the analysis techniques. We start by outlining several representative systems that use resource bounds to improve coordination: design-time cost analysis through the use of structured program notations such as

the Bird-Meertens-Formalism (BMF) or Bulk Synchronous Processes (BSP); compile-time cost analysis through the use of type inference, abstract interpretation, or constraint system solving; run-time cost analysis through the (abstract) execution of (abstracted) input based on a costed semantics.

Section 4 outlines six applications of resource analysis techniques in a parallel/distributed context. The applications are selected to be effective, i.e. they improve parallel/distributed coordination, and representative. The applications are representative in utilising a range of analyses from the most abstract to the most concrete, and in using the resource information for a range of coordination purposes including resource-safe execution, compiler optimisations, optimising parallel execution and enabling mobility.

Section 5 investigates in the impact of the choices made for the cost model and analysis technique on the concrete application domain. Section 6 summarises the general principles governing why combinations of cost model and analysis technique are effective, and speculates on future trends for resource analysis in the parallel/distributed arena.

2 Coordination Cost Models

Coordination cost models provide tractable abstractions of the performance of real parallel or distributed systems. They cover a well populated continuum from the most simple and abstract, through to the highly detailed and concrete. A detailed survey of parallel cost models is given in [25]. For our purposes we note four well-known and representative examples that are used in the analysis methodologies presented subsequently. As we shall see in Section 4, classical sequential cost models are also useful, e.g. using predicted execution time for tasks to inform scheduling.

2.1 PRAM

The Parallel Random Access Machine (PRAM) model [12] is the most abstract parallel cost model. PRAM is the fundamental parallel machine and cost model within the parallel algorithms and complexity research community. In its simplest form, it models stepwise synchronous, but otherwise unrestricted access to a shared memory by a set of conventional sequential processors. At each synchronous step, each processor performs one operation from a simple, conventional instruction set. Each such step is costed at unit time, irrespective of the operations involved, and in particular, irrespective 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 propose that each processor first sums a distinct sub-array of size $\frac{n}{p}$ items, then a single processor sums the $\frac{n}{p}$ partial results of the first phase. An informal PRAM cost analysis would capture the cost of this algorithm as

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

A more sophisticated algorithm B might suggest summing the partial results in a parallel tree-like structure (usually known as "parallel reduction"). The corresponding analysis would suggest a run-time of

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

The designer is thus alerted to the coarse-grain observation that the more sophisticated algorithm may be asymptotically faster, but only for large p.

The PRAM model ignores a number of important issues such as contention, memory hierarchy, underlying communication infrastructure and all processor internal issues. Nevertheless, it has provided a durable and sound basis for at least the initial phases of asymptotically analysed parallel algorithm design. A plethora of variants aim to introduce more pragmatic cost issues. For example, the EREW-PRAM disallows steps in which any shared memory location is accessed by more than on processor (algorithms A and B both satisfy this requirement). In contrast, the CRCW-PRAM removes this restriction, with sub-variants defining behaviour when clashes occur.

2.2 BSP

The Bulk Synchronous Parallel (BSP) model [4] occupies a less abstract position in the cost model spectrum than PRAM. In contrast to PRAM it recognises that synchronisation is not free, that sharing of data involves communication (whether explicitly or implicitly), and that the cost of this communication, both absolutely and relative to that of processor-local computation can be highly machine dependent. To tame this complexity BSP introduces a constrained operational model. Thus, a BSP computer consists of processors connected by a communication network. Each processor has a fast local memory, and may follow its own thread of computation.

A BSP computation proceeds in a series of *supersteps* comprising three stages. *Independent concurrent computation* on each processor using only local values. *Communication* in which each processor exchanges data with every other. *Barrier synchronisation* where all processes wait until all other processes have finished their communication actions.

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 sum of the cost of the longest running local computation, the cost of the maximum communication between the processes, and the cost of the barrier synchronisation. The costs are computed in terms of three abstract parameters which 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 above, translated for BSP, would have asymptotic run-times of

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

with the first two terms contributed by computation, the third by communication and the fourth by the need for two supersteps, and

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

with the first time corresponding to local computation, and the other three terms to computation, communication and synchronisation summed across the $\log p$ supersteps of the tree-reduction phase. The analysis reveals the vulnerability of the algorithm B to architectures with expensive synchronisation.

The constrained computational model allows BSP implementations to provide a benchmark suite which derives concrete, machine-specific values for g and L. These can then be inserted into the abstract (architecture independent) cost already derived for a given program, to predict its actual run-time.

While BSP makes no attempt to account for processor internals or memory hierarchy (other than indirectly through benchmarking) or specific communication patterns (indeed, classical BSP implementations rely on randomisation to deliberately *obliterate* patterns in the interests of predictability), a considerable literature testifies to the pragmatic success of the approach [4].

2.3 Cost Semantics

Cost semantics occupy a more concrete position in the cost model spectrum than BSP. A cost semantics is a non-standard semantics defining computational costs, rather than result values, for program expressions. To this end costs are assigned to primitive operations, and language constructs combine the costs of their subterms, e.g. the cost of a adding two values in the array sum example above might be one plus the cost of evaluating the left and right operands. The cost semantics thus defines the exact costs incurred when running the program. In general, this requires knowledge of run-time values. The role of compile-time analyses, as discussed in the next section, is to abstract over run-time values and deliver a static approximation of this value. The costs of coordination operations like communication latency, or synchronisation, are accounted as some architecture dependent value, possibly using some coordination cost model.

Concrete cost semantics aim to provide precise cost estimates by assigning accurate costs to terms, often based on careful profiling of a target architecture [5] or by explicitly modelling low-level architecture aspects such as the contents of the cache. In contrast an *abstract* cost semantics, as the one sketched above, assigns unit costs to terms and hence is both simpler and architecture independent.

2.4 Accurate Hardware Models

Accurate hardware models occupy the most concrete position in the cost model spectrum. These models provide precise cost information of low-level code, for example by providing time information in clock cycles for each machine instruction. Such level of detail is required for industry strength worst-case execution

time (WCET) analyses. These analyses must be safe in the sense of always producing upper bounds. They also have to be precise to be of any practical use. One example of a WCET analysis that combines these features is AbsInt's aiT tool for sequential code [11].

3 Resource Analyses

Resource analysis has a role to play in each of the three phases of a program's lifetime: design, compilation and execution (run-time).

3.1 Design Time Analysis

Abstract cost models based around PRAM, BSP and Bird-Meertens-Formalism (BMF) enable the programmer to reason about costs during program design, as shown by the example in the previous section. The models often require that the program is expressed using a specific structure, e.g. as a sequence of supersteps for BSP analysis. A significant advantage of these techniques is that, guided by the model, the programmer can relatively cheaply transform the program design to reduce the consumption of a specific resource, before committing to an implementation.

3.2 Compile Time Analysis

In the area of compile time analyses many techniques have been developed to statically infer information of the behaviour of the program at runtime. The best known techniques are type inference, abstract interpretation, and constraint system solving, and they may be used in combination.

Type Inference: Based on the observation that type inference can be separated into two phases, collecting constraints on type/resource variables and solving these constraints [22], several type-inference based analyses have been developed that extend Hindley-Milner type inference to collect and solve constraints on resources.

Abstract Interpretation: Abstract interpretation [7] defines an abstract domain of values and executes the program on this domain. Functions are mapped to abstracted functions that operate over the abstract domain. The analysis then proceeds by executing these abstract functions, and in particular finding fixpoints for recursive functions. Traditionally, this is used to infer qualitative information. By using a richer abstract domain quantitative information can be modelled, too. Many practically useful techniques have been developed for this process, and therefore well-developed inference engines exists that can be used for resource analysis.

Constraint System Solving: This approach is related to the type inference approach. In the former, constraints are collected during type inference and then solved separately. In the (pure) constraint system solving approach the collection of constraints is not tied to type inference. An example of this approach is control flow analysis [30].

Examples: A concrete example of an analysis based on abstract interpretation is AbsInt's aiT tool. It operates on machine code for a range of embedded systems processors, and produces sequential worst case execution (WCET) bounds. In order to obtain accurate bounds, the analysis models details of the hardware architecture, in particular the cache behaviour and the possible pipeline structure of the processor. Another analysis using abstract interpretation is the COSTA [1] system. It is generic in the definition of "resources", 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 run-time, profiling approach is presented in [26].

3.3 Run-Time Analysis

Run-time cost analysis typically entails the abstract execution with some abstracted input. It differs from profiling in that the execution and resources are abstract rather than real. It is often used in conjunction with a static resource analysis, for example to approximate the sizes of key data structures, e.g. [17, 26].

4 Parallel/Distributed Resource Analysis Applications

This section outlines six representative parallel/distributed applications of resource analysis techniques. The applications are ordered from those applying the most abstract analysis (BMF-PRAM) to the most concrete (Type-based Analysis with a Precise Model).

It is well known that performance analysis within conventional programming models is undecidable. Pragmatic progress can be made by relaxing the extent to which we hope for computable analysis for example by requiring oracular insight from the programmer and/or by constraining the programming model in some way.

For each system we outline how resource information is obtained and applied. Each representative model is effective, i.e. the cost information improves coordination and hence performance.

4.1 BMF-PRAM

The Bird-Meertens Formalism (BMF) [2] 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. While independent of

any explicit reference to parallelism, many of its operations clearly admit potentially efficient parallel implementation. A number of projects have attempted to exploit this opportunity by adding parallel cost analyses to BMF inspired programming models.

In [31] 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 (for example, 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.

In [19], Jay et al. build a formal cost calculus for a small BMF-like language using PRAM as the underlying cost model. In order to aid implementation, 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, though structurally constrained source language, being entirely static in nature and with varying degrees of formality and support.

4.2 BMF-BSP

Building on Jay and Skillicorn's seminal work, a number of projects have sought to inject more realism into the costing of BMF inspired parallel programming frameworks. The primary vehicle to this end was the substitution of BSP for PRAM as the foundational cost model [19, 15]. In particular, [15] defines and implements a BMF-BSP calculus and compares the accuracy of its predictions with the runtime of equivalent (but hand-translated) BSP programs. Using 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.

Meanwhile, in a more informal setting reflecting the approach of [31], [3] reports upon 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 (and for the first time, experimentally validated), while retaining similarly structured models and support. As a by-product of the use of BSP, analyses are now target architecture specific, once instantiated with the machine's BSP constants, though still static with respect to any particular instance.

4.3 Skeleton-based Approaches

The *skeleton* based approach to parallel programming [6] advocates that commonly occurring patterns of parallel computation and interaction be abstracted as library or language constructs. These may be tied to data-parallel bulk operations, in the style of BMF, or used to capture more task oriented process networks (for example, pipelines). Various projects have sought to tie cost models to skeletons and to use these, either explicitly or implicitly to guide program development and implementation.

For example, based around a simple model of message passing costs, [14] uses meta-programming to build cost equations for a variety of skeleton implementations into an Eden skeleton library, allowing the most appropriate implementation to be chosen at compile-time given instantiation of target machine specific parameters (i.e. in the style of, but distinct in detail from, BSP). Discrimination between four possible variants of a farm skeleton, used to implement a Mandelbrot visualisation is reported.

Meanwhile, [13] describes an attempt to embed the BSP model directly into a functional programming language, ML. At the level of parallelism, the programming model is thus constrained to follow the BSP superstep constraints (which might be viewed as relatively loose skeleton), while 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.

The approach proposed by [32] presents the programmer with imperative skeletons, each with an associated parallel cost model. The models are defined in a performance enhanced process algebra [16], parameterised by a small number constants derived by running benchmark code fragments. As in [14] models of competing implementation strategies are evaluated and the best 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 actual performance. Where a significant discrepancy is found, the computation can be halted, re-evaluated and rescheduled.

These approaches are strong in terms of language support, offering essentially a two-layer model in which parallelism is constrained by the available skeleton functions but local computation is free and powerful. The cost foundations are of middling accuracy, sometimes augmented by the use of real code profiling. They employ a range of static and dynamic analysis.

4.4 Using Statically Inferred Granularity Information for Parallel Coordination

This section outlines several systems that apply cost information in the context of parallel computation to decide whether a parallel thread should be generated for some computation. In particular, it should become possible to identify very small pieces of computation, for which the overhead of generating a parallel

thread is higher than the actual computation itself. Hence the characteristic feature of the cost information here is that while it must be accurate for small computations, it can be far less accurate for larger computations. Potentially all computations beyond a certain threshold can be mapped to an abstract value of infinity.

Static Dependent Costs: Reistad and Gifford [29] define the notion of static dependent costs for the analysis of a strict, higher-order functional language with imperative features. These costs describe the execution time of a function in terms of the size of its input by attaching cost information to the type of a function. Thereby it becomes possible to propagate cost information from the definition of a function to its use, enabling the static, type-based analysis of higher-order programs. The static inference of cost expressions is combined with runtime calculation that instantiate cost expressions for concrete sizes to gain concrete estimates. Runtime measurements of the system show that their cost estimates are usually within a factor of three of the real costs. This information is used in a dynamic profitability analysis, that compares the cost estimate of an expression with the thread creation overhead, and generates parallelism only if it is profitable. A game of life program, based on a parallel map operation exploiting this profitability analysis, achieved a speedup of more than two compared to a naive version of a parallel map on a four processor SGI shared-memory machine.

Dynamic Granularity Estimation: Another instance of this approach is [17], where a technique of dynamic granularity estimation for strict, list-based, higher-order languages is developed. This technique comprises two components: A compile-time (static) component, based on abstract interpretation to identify components whose complexity depends on the size of a data structure; and a run-time (dynamic) component, for approximating sizes of the data structures at run-time. Based on the results of the static component, the compiler inserts code for checking the size of parameters at certain points. At runtime the result of these checks determine whether a parallel task is created or not. The dynamic component is implemented on a Sequent Symmetry shared-memory machine on top of a parallel SML/NJ implementation. It is stated that the runtime overhead for keeping track of approximations (one additional word per cons cell) is very low. For the quicksort example an efficiency improvement of 23% has been reported.

Sized Time Systems: The sized time system in [24] develops a type-based inference of execution time and combines it with sized types [18], a static type system for inferring bounds on the size of data structures. Thus, in contrast to the previous systems, no run-time analysis is required. As in the previous systems, the analysis of time information is restricted to non-recursive functions. As traditional, the inference is separated into a phase of collecting constraints, inequalities over natural numbers, and a separate phase of solving these constraints. A simulator for the parallel execution of Haskell programs has been used to implement several scheduling algorithms that make use of granularity information. In

its most naive instance all potential parallelism below a fixed granularity threshold is discarded. In a second variant, granularity information is used to always pick the largest item when generating new parallelism. In a final version, granularity information is used by the scheduler to favour the largest thread upon re-scheduling. The results with these three version showed [23][Chapter 5], that the most naive version of a fixed granularity threshold performed best, since the management overhead of the more refined policies dominated the gains in execution time.

In summary, all three *systems* discussed here are based on an abstract, architecture-independent cost model, and use a static, type-based cost analysis to determine size-dependent bounds. Two of the three systems combine these with a very simple run-time analysis, which mainly supplies size information. The *languages* covered are predominantly functional, although the static dependent cost system also covers imperative constructs. The *run-time techniques* that use the provided cost information are very simple: in most cases a binary decision on the profitability of a potential parallel thread is made. Arguably the use of the cost information is *a priori* limited by the choice of an abstract cost model, which cannot provide precise bounds. However, measurements of the system show that even with the abstract cost model, cost predictions, where possible, are reasonably accurate.

4.5 Abstract Cost Semantics: Autonomous Mobile Programs

Autonomous mobile programs (AMPs) periodically use a cost model to decide where to execute in a network [9]. The key decision is whether the predicted time to complete on the current location is greater than the time to communicate to the best available location and complete there.

The AMP cost model is an abstract cost semantics for a core subset of the Jocaml mobile programming language including iterating higher-order functions like `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.

The AMP continuation cost model is generated statically, and is then parameterised dynamically to determine movement behaviour. Key dynamic parameters include the current input size, execution speed on the current location, predicted execution speeds of alternative locations.

In summary the AMP abstract costed operational semantics is applied to a core mobile functional language with higher-order functions. The model is statically generated but dynamically parameterised. While such an abstract cost semantics provides low accuracy, empirical results show that the information adequately informs mobility decisions [9].

4.6 Type-based Analysis (Precise Model): Resource-safe Execution in Hume

The goal of resource-safe execution is to statically guarantee that available resources are never exhausted. This is highly desirable in many contexts, e.g. to provide resource guarantees for mobile code, or in embedded systems where resources are highly constrained.

With multi-core architectures entering the main-stream of computer architectures, embedded system designers are looking into exploiting the parallelism provided on such platforms. Thus, the new challenge is to combine resource-safe execution with a model for parallel execution that can effectively, and safely exploit the parallelism. One aspect to this challenge is to best use the special nature of the resource bounds, required for resource-safe execution, to guide parallel execution.

In order to meet safety requirements on embedded systems, the resource predictions, and hence the cost model, have to be *upper bounds* on the concrete resource consumption rather than simple predictions. These bounds don't necessarily have to be precise, however they must be concrete enough to assure that no concrete evaluation exceeds them. Furthermore, formal guarantees of the validity for these bounds are highly desirable.

The resource analysis for Hume [20], 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. It produces, where possible, linear bounds on the resource consumption. Some supported resources are heap- and stack-space consumption, and worst case execution time.

The underlying cost model is an accurate hardware model obtained by performing machine-code-level worst-case execution time analysis on the operations of the underlying abstract machine. Thus, it is a *concrete cost model*, taking into account hardware characteristics such as cache and pipeline structure. It is a *safe cost model* in the sense that all costs are upper bounds. The results of the resource analyses for space and time have been validated against measurements obtained on real embedded systems hardware for a range of applications [21].

The Hume compiler currently uses the resource information only in determining sizes of buffers etc, needed to assure resource-safe, single processor execution. In the longer term this information will also be used in other components of the system, for example in the scheduler on the box layer. The decomposition of the program into boxes provides a natural model for parallel execution on multi-core machines. In this context, the number of threads, namely boxes, is statically fixed. The main usage of the resource information is therefore in statically mapping the threads to the available cores and in dynamically deciding which thread to execute next. Since on box layer the execution of a program is an alternating

sequence of compute- and communicate-steps, the mapping process is akin to the process of developing a parallel program in a BSP model.

5 Cost Model & Analysis Critique

Given the spectrum of cost models and associated analysis techniques, what combination should be selected for a specific parallel or distributed application? This section investigates why a specific cost model and analysis technique proves effective in the specific parallel/distributed context.

5.1 BMF-PRAM

In common with their PRAM base, the techniques discussed in Section 4.1 are most appropriate in the early phases of algorithm design, rather than detailed program development. The techniques enable the designer to quickly compare coarse performance estimates of alternative approaches. An informal, even asymptotic flavour predominates.

5.2 BMF-BSP

The BMF-BSP approaches discussed in Section 4.2 are more appropriate when a reasonably detailed algorithm already exists, allowing more refined, machine-sensitive cost modelling as a concrete program is refined. They are most appropriate in (indeed, almost constrained to) contexts which provide a BSP library implementation, with its associated benchmark suite.

5.3 Skeleton-based Approaches

Since the skeleton techniques outlined in Section 4.3 largely aim to absolve the programmer of responsibility for the detailed expression and exploitation of parallelism, resource analysis techniques are typically exploited in the library implementation itself, both statically and even dynamically. With the exception of the work in [13], the programmer is unaware of the cost model's existence.

5.4 Using Statically Inferred Granularity Information for Parallel Coordination

The three granularity estimation systems outlined in Section 4.4 share the following notable features. The inference engine is simple and cheap, but limited to non-recursive functions. Most of the information is inferred statically, but in some cases a light-weight run-time analysis is applied, too. The inferred quantitative information is mostly used in a qualitative way (whether a thread is profitable, i.e. large enough to be evaluated in parallel).
For an application domain where imprecise, mostly qualitative information is sought, *ad hoc* techniques or light-weight, type-inference based techniques work very well. The mostly static nature of the analysis avoids run-time overhead.

5.5 Abstract Cost Semantics: Autonomous Mobile Programs

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, i.e. parameterising the model with current performance.

5.6 Type-based Analysis (Precise Model): Resource-safe Execution in Hume

The following characteristics of the resource analysis for Hume 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. To deliver such guarantees, the type-based analysis builds on strong formal foundations and the type system is proven sound. Through its tight integration of resource information into the type system, 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. To facilitate tight upper bounds 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 re-targeted for other (quantitative) resources. By using a standard linear program solver in the constraint solving stage, we achieve an efficient analysis.

6 Discussion

A key purpose of this paper is to inform the resource analysis community of the diversity and widespread usage of resource analysis in the parallel and distributed community, and what constitute effective analyses for the community. We have outlined a continuum of coordination cost models and a range of analysis techniques for parallel/distributed programs. By critiquing six representative resource analysis applications we identify the following general principles governing why the combination of techniques is effective in its context.

- Predominantly, the effective parallel/distributed resource analyses have been carefully designed to deliver the right information for the specific coordination purposes, and this has a number of aspects.
  - The analysis must deliver information that is sufficiently accurate. Often a surprisingly simple cost model is effective, e.g. for the AMPs in Section 5.5.

    - The analysis must combine static and dynamic components appropriately. For some applications purely static information suffices, where others require at least some dynamic information (Section 4.4).
    - In many cases it is sufficient for the analysis to produce qualitative predictions, e.g. is it worth creating a thread to evaluate an expression. However in some scenarios, such as resource-safe execution, the analysis must produce (upper) bounds (Section 4.6).
- Highly abstract resource analyses like BMF-PRAM are informative even at early phases of parallel algorithm design (Section 5.1).
- More refined, architecture dependant analyses can be utilised during parallel program development (Section 5.2).
- Improving reusable coordination abstractions like algorithmic skeletons can have a significant impact and resource analyses are commonly applied within skeleton libraries (Section 5.3).
- Even partial cost information can prove very useful, for example in deciding whether to generate parallelism (Section 4.4).
- Often the inferred quantitative information is mostly used in a qualitative way. Therefore, imprecise or relative resource information is sufficient (Section 4.4).

Clearly resource analysis will remain an important tool for parallel/distributed systems, and we trust that the principles above will assist in the design of future systems. We anticipate that these systems will be able to exploit the rapidly-improving resource analysis technologies. Indeed recent advances have already widened the range of programs for which static information can be provided (a detailed survey of WCET analyses is given in [10]) and caused a shift from run-time to compile-time techniques (Section 3). Some important trends that we anticipate in the near future are as follows.

In the area of static analyses there is a general trend to type-based analysis and to enriching basic type systems with information on resource consumption. The standard type-inference machinery has proven to be a very flexible engine that can be re-used to infer resource information.

Static analyses are getting increasingly complex and therefore more error-prone. At the same time automated theorem proving techniques increasingly mature. The combination of both, for example through formalised soundness proofs of the analysis, is desirable in particular in safety-critical systems. Alternatively, proof-carrying-code [27] or abstraction carrying code, avoid the (complex) soundness proof in general, and perform (formal) certificate validation on each program instead.

Hardware, and hence precise cost models, are becoming increasingly complex. This will push existing, low-level resource analysis to their limits and significantly worsen the WCET bounds that are achievable. For these reasons, probabilistic cost models are of increasing interest to the WCET community [28]. In the context of parallel and distributed execution, where predictions rather than bounds are sufficient, this trend will be even more relevant, but is currently not explored.

With respect to programming models, increasing interest in structured and constrained approaches [8] can be seen to bring benefits in terms of simplification,

when coupled with correspondingly structured cost models. Constraining the patterns of parallelism available to the programmer facilitates the construction of tractable cost models, parameterised by the costs of the composed sequential fragments.

References

1. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. COSTA: Design and Implementation of a Cost and Termination Analyzer for Java Bytecode. In *Intl. Symp. on Formal Methods for Components and Objects (FMCO'07)*, LNCS 5382, pages 113–132, Amsterdam, The Netherlands, October 24–26, 2007. Springer.
2. R. Bird and O. de Moor. *Algebra of Programming.* Prentice Hall, 1997.
3. H. Bischof, S. Gorlatch, and E. Kitzelmann. Cost Optimality and Predictability of Parallel Programming with Skeletons. In *Euro-Par 2003 Parallel Processing*, LNCS 2790, pages 682–693. Springer-Verlag, 2003.
4. Rob Bisseling. *Parallel Scientific Computation: A Structured Approach using BSP and MPI.* Oxford University Press, 2004.
5. A. Bonenfant, C. Ferdinand, K. Hammond, and R. Heckmann. Worst-Case Execution Times for a Purely Functional Language. In *Implementation of Functional Languages (IFL 2006)*, LNCS 4449, pages 235–252, Budapest, Hungary, September 4–6, 2006. Springer.
6. Murray Cole. *Algorithmic skeletons: structured management of parallel computation.* MIT Press, 1989.
7. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL'77 — Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, January 1977.
8. Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.
9. Xiao Yan Deng, Phil Trinder, and Greg Michaelson. Cost-Driven Autonomous Mobility. *Computer Languages, Systems and Structures*, 36(1):34–51, 2010.
10. R. Wilhelm *et al.* The Worst-Case Execution-Time Problem—Overview of Methods and Survey of Tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):1–53, 2008.
11. C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise WCET determination for a real-life processor. In *Intl Workshop on Embedded Software (EMSOFT'01)*, LNCS 2211, pages 469–485, Tahoe City, USA, October 8–10, 2001. Springer.
12. Steven Fortune and James Wyllie. Parallelism in random access machines. In *STOC '78: Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 114–118. ACM, 1978.
13. Frédéric Gava. Bsp functional programming: Examples of a cost based methodology. In *ICCS '08: Proceedings of the 8th international conference on Computational Science, Part I*, pages 375–385. Springer-Verlag, 2008.
14. K. Hammond, R. Loogen, and J. Berhold. Automatic Skeletons in Template Haskell. In *Proceedings of 2003 Workshop on High Level Parallel Programming, Paris, France*, June 2003.
15. Yasushi Hayashi and Murray Cole. Automated cost analysis of a parallel maximum segment sum program derivation. *Parallel Processing Letters*, 12(1):95–111, 2002.

16. Jane Hillston. *A compositional approach to performance modelling.* Cambridge University Press, New York, NY, USA, 1996.
17. Lorenz Huelsbergen, James R. Larus, and Alexander Aiken. 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*, pages 79–90, New York, NY, USA, 1994. ACM.
18. R.J.M. Hughes, L. Pareto, and A. Sabry. Proving the Correctness of Reactive Systems Using Sized Types. In *POPL'96 — Symposium on Principles of Programming Languages*, St. Petersburg Beach, Florida, January 1996. ACM.
19. C. B. Jay, M. I. Cole, M. Sekanina, and P. Steckler. A monadic calculus for parallel costing of a functional language of arrays. In *Euro-Par'97 Parallel Processing, volume 1300 of Lecture Notes in Computer Science*, pages 650–661. Springer, 1997.
20. S. Jost, H-W. Loidl, K. Hammond, N. Scaife, and M. Hofmann. "Carbon Credits" for Resource-Bounded Computations using Amortised Analysis. In *Intl. Symp. on Formal Methods (FM '09)*, LNCS. Springer, November 2009. to appear.
21. S. Jost, H-W. Loidl, N. Scaife, K. Hammond, G. Michaelson, and M. Hofmann. Worst-Case Execution Time Analysis through Types. In *Proc. of Euromicro Conference on Real-Time Systems (ECRTS'09)*, pages 13–17, Dublin, Ireland, July 1–3, 2009. Work-in-Progress Session.
22. T-M. Kuo and P. Mishra. Strictness Analysis: a New Perspective Based on Type Inference. In *FPCA'89 — Conference on Functional Programming Languages and Computer Architecture*, pages 260–272, Imperial College, London, UK, September 11–13, 1989. ACM Press.
23. H-W. Loidl. *Granularity in Large-Scale Parallel Functional Programming.* PhD thesis, Department of Computing Science, University of Glasgow, March 1998.
24. H-W. Loidl and K. Hammond. A Sized Time System for a Parallel Functional Language. In *Glasgow Workshop on Functional Programming*, Ullapool, Scotland, July 8–10, 1996.
25. B.M Maggs, L.R. Matheson, and R.E. Tarjan. Models of Parallel Computation: A Survey and Synthesis. In *HICSS'95: Proceedings of the 28th Hawaii International Conference on System Sciences*, pages 61–70, Washington, DC, USA, 1995. IEEE Computer Society.
26. E. Mera, P. López-García, G. Puebla, M. Carro, and M. Hermenegildo. Combining Static Analysis and Profiling for Estimating Execution Times. In *Intl. Symp. on Practical Aspects of Declarative Languages (PADL'07)*, LNCS 4354, pages 140–154. Springer, January 2007.
27. G. Necula. Proof-carrying-code. In *Symposium on Principles of Programming Languages (POPL'97)*, pages 106–116, Paris, France, January 15–17, 1997.
28. E. Quiñnones, E.D. Berger, G. Bernat, and F.J. Cazorla. Using Randomized Caches in Probabilistic Real-Time Systems. In *Proc. of Euromicro Conference on Real-Time Systems (ECRTS'09)*, pages 129–138, Dublin, Ireland, July 1–3, 2009. IEEE.
29. Brian Reistad and David K. Gifford. Static dependent costs for estimating execution time. In *LFP '94: Proceedings of the 1994 ACM conference on LISP and functional programming*, pages 65–78, New York, NY, USA, 1994. ACM.
30. Olin Shivers. *Control-Flow Analysis of Higher-Order Languages.* PhD thesis, School of Computer Science, Carnegie Mellon University, May 1991.
31. D. B. Skillicorn and W. Cai. A cost calculus for parallel functional programming. *J. Parallel Distrib. Comput.*, 28(1):65–83, 1995.
32. Gagarine Yaikhom, Murray Cole, and Stephen Gilmore. Combining measurement and stochastic modelling to enhance scheduling decisions for a parallel mean value

analysis algorithm. In *International Conference on Computational Science (2)*, pages 929–936, 2006.