Program Execution on Reconfigurable Multicore Architectures

Sanjiva Prasad
Indian Institute of Technology Delhi
New Delhi, India
sanjiva@cse.iitd.ac.in

Based on the two observations that diverse applications perform better on different multicore architectures, and that different phases of an application may have vastly different resource requirements, Pal *et al.* proposed a novel reconfigurable hardware approach for executing multithreaded programs. Instead of mapping a concurrent program to a fixed architecture, the architecture adaptively reconfigures itself to meet the application's concurrency and communication requirements, yielding significant improvements in performance. Based on our earlier abstract operational framework for multicore execution with hierarchical memory structures, we describe execution of multithreaded programs on reconfigurable architectures that support a variety of clustered configurations. Such reconfiguration may not preserve the semantics of programs due to the possible introduction of race conditions arising from concurrent accesses to shared memory by threads running on the different cores. We present an intuitive partial ordering notion on the cluster configurations, and show that the semantics of multithreaded programs is always preserved for reconfigurations "upward" in that ordering, whereas semantics preservation for arbitrary reconfigurations can be guaranteed for well-synchronised programs. We further show that a simple approximate notion of efficiency of execution on the different configurations can be obtained using the notion of amortised bisimulations, and extend it to dynamic reconfiguration.

1 Introduction

The traditional approach to multiprocessing is to map multithreaded applications or multiprogram workloads onto a chosen multicore architecture. Pal *et al.* showed that due to the diversity of software applications, this "one architecture fits all" approach often yields sub-optimal performance [7]. For instance, programs with mostly independent threads having little communication perform well on symmetric multiprocessors (SMP) whereas those with more communication and shared variables perform better on chip multiprocessors (CMP). Indeed, even a single application can exhibit vastly diverse resource requirements during different phases of its execution. Accordingly, those authors identified different reconfiguration parameters (e.g., number of cores, cache size and cache sharing) and proposed a reconfigurable multicore tile-based architecture which supports dynamic *adaptability of the multicore hardware to the software's resource requirements* [8], obtaining significant performance improvements. The *hardware morphs itself* to a configuration that delivers better performance for that particular phase of program execution (using heuristics to detect such phase changes). The overhead for reconfiguration is usually significantly lower than the performance benefits. However, it is not entirely obvious whether such a dynamic reconfiguration preserves the intended semantics of the application with respect to a reference architecture, nor is there a theoretical framework for comparing the performance benefits. We believe there should be a formal basis for dynamically reconfigurable multiprocessors, which constitute an innovative technology trend.

D. Orchard and N. Yoshida (Eds.): Programming Language Approaches to Concurrency- and Communication-Centric Software (PLACES 2016).
EPTCS 211, 2016, pp. 83–91, doi:10.4204/EPTCS.211.9

In this paper, we present an operational account of execution of multithreaded programs on *dynamically reconfigurable* multicore architectures, which to our knowledge is new. Examining how a variety of *cluster-based* architectural configurations partition cores and share cache, we see that partition refinements provide a natural partial ordering on cluster configurations. We also find that instead of requiring different operational formulations for each configuration, our earlier work [2] provides a *uniform* abstract operational semantics, whence we can both compare execution behaviour semantics as well as express dynamic reconfigurability. Leveraging results about abstract cache models from our work [2], which followed the seminal approach of Boudol and Petri [1] of showing that "well-synchronised programs have the same semantics in relaxed memory models as on sequentially consistent models", we show that dynamic reconfiguration is semantics-preserving for such data-race-free programs. Further, by associating an approximate cost with each operation, we adapt Kiehn and Arun-Kumar's notion of *amortised bisimulation* [3] to obtain a framework for comparing performance on the various architectural configurations and reasoning about the benefits of dynamic reconfiguration.

It should be clarified at the outset that we are presenting execution semantics at the *architectural* level, *below the program or OS level* where threads are assigned to cores. Reconfiguration happens dynamically during program execution, and is *outside program control*. In this respect, this work differs from that of Krishnan [4, 5] and also the large body of work related to assigning threads to cores (which has anyway become a less pressing issue in multicore systems). For simplicity, we confine our study to homogeneous core architectures, and the reconfiguration parameters to core clustering and cache fusion/splitting, and do not consider other parameters such as core fusion/splitting, core allocation, management of power and clocking, and cache allocation.

2 The Reference Model

The reference architectural model with respect to which we compare the semantics of program execution consists of a collection of cores connected via a bus to a single shared memory module. Under the assumption of having the requisite number of cores, this reference model will exhibit behaviours consistent with *pomset* semantics [9] of multiple sequential processes with a *sequentially consistent* shared memory [6]. In a sequentially consistent memory model, writes and reads are atomic operations, and occur in program order within a thread.

Execution states are written as $\mathbb{C} = (S,P)$, where S is the shared *store* and P is a vector of threads. For simplicity, we assume each thread runs on its own core, with P_i denoting the i^{th} thread. The operational semantics are given in Figure 1. Since our focus is on the observable actions on shared memory at the architectural level, only the transitions relating to reading and writing from memory are shown, eliding transitions for instructions not involving the store. Since the bus enforces *mutually exclusive access* to the memory module, we adopt an interleaving view of execution. An obvious alternative approach would have been to consider *synchronous transitions*, labelled by *vectors of actions*, the components of which are contributed by each core. However, since certain cores may idle (e.g., for power efficiency reasons) and since we are not making any assumptions about clock synchronicity, that approach would not be appropriate.

Transitions related to accessing the store are of the form $\xrightarrow{(a,i)}$, where i is used to indicate that the transition is for thread P_i (or core i), and a denotes the action. The possible actions are: $\mathtt{rd}_x^v$ (the value v is read from variable x) and $\mathtt{wr}_x^v$ (the value v is written to the variable x), apart from the reductions (labelled τ) not involving the store. When we do not care what value was read/written, we use $\mathtt{rd}_x$ and $\mathtt{wr}_x$. We associate an approximate *cost* δ for accessing the store, with $c(\mathtt{rd}_x^v) \simeq c(\mathtt{wr}_x^v) \simeq \delta$. For simplicity, we

$$(read) \quad (S, P_i[x]) \xrightarrow{(\mathtt{rd}_x^v, i)}_S (S, P_i[v]) \quad \text{where } S(x) = v$$
$$(write) \quad (S, P_i[x{:=}v]) \xrightarrow{(\mathtt{wr}_x^v, i)}_S (S[x \leftarrow v], P_i[()])$$

Figure 1: Specification semantics for read and write in the reference architecture

ascribe a uniform cost θ for τ-labelled transitions (typically $\theta < \delta$)[1]

In a sequence of transitions $\mathbb{C}_0 \xrightarrow{(a_0,i_0)} \cdots \xrightarrow{(a_n,i_n)} \mathbb{C}_{n+1}$, two *concurrently enabled but conflicting* transitions $\xrightarrow{(a_j,i_j)}$ and $\xrightarrow{(a_k,i_k)}$ are said to form a *race* (on variable x) if $i_j \neq i_k$ and $a_j, a_k \in \{\mathtt{rd}_x, \mathtt{wr}_x\}$ and at least one is $\mathtt{wr}_x$. Races make computational results dependent on scheduling decisions, and as a consequence programmers use synchronisation mechanisms such as locks, barriers, fences, etc. to avoid the occurrence of such data races.

3 Implementation Models

There is a variety of configurations for multicore architectures. Common among them are chip multiprocessors (CMP) and symmetric multiprocessing (SMP). A major difference between them lies in the organisation of their cache hierarchy. Caches are important architectural features that significantly speed up execution of programs by exploiting locality of memory accesses and reducing their latency.

In a CMP configuration, each core possesses its private data and instruction $L1$ caches, but several cores share a common $L2$ cache, which lies above the slower main memory. In contrast, in SMP, the $L2$ cache is also private to each core. These two configurations may be considered the extremes of a range of *clustered configurations* or *clusterings*, where the multicore system consists of a collection of clustered cores. Within a cluster, each core possesses its private $L1$ data and instruction caches, but the cores share a $L2$ cache. Thus, SMP is the case where the cluster size is 1, whereas CMP puts all the cores in one cluster. Pal *et al.* use the notation $k(c_1, \ldots, c_k)$, where $(\Sigma_{j=1}^{k} c_j) = N$ to describe a system of N cores configured into k clusters, where the j^{th} cluster has c_j cores. SMP is therefore written as $N(1, \ldots, 1)$ while CMP is represented as $1(N)$. For simplicity, we will only concern ourselves with a memory structure consisting of a $L2$ caches and shared main store. A more detailed model can address the similar issues that arise in the treatment of $L1$ *vis à vis* $L2$ caches and main memory.

We observe that a clustering represents a *partition* of cores; given a clustering Q, we write $i \sim_Q j$ if cores i and j are in the same cluster. Thus in SMP, the equivalence classes are singletons, whereas in pure CMP, all cores are in the same equivalence class. If clustering Q is a *partition refinement* of Q', we write $Q \leq Q'$.[2] Note that if $i \sim_Q j$, then i, j share the same $L2$ cache.

3.1 Implementation semantics

We now refine the reference model by introducing caches into the architecture. The store component is replaced by a tuple (S, C), where S is the store (as earlier) and C is a vector of caches. The caches contain a local copy of a subset of the store. Due to differences in the local caches, each core has a potentially different view of the memory. C_i denotes the ($L2$) cache available to core i. In clustering Q, if $i \sim_Q j$, then C_j and C_i are the same and so have the same contents.

[1] Assumptions on θ do not have any significance in this paper.

[2] Note that refinements are lower in the ordering.

$$(\mathit{LocalRead}) \quad (S,C,P_i[x]) \xrightarrow{(\mathtt{rdl}^v_x,i)}_Q (S,C,P_i[v]) \qquad \text{where } x \in \mathit{dom}(C_i) \wedge C_i[x].\mathit{val} = v$$

$$(\mathit{StoreRead}) \quad (S,C,P_i[x]) \xrightarrow{(\mathtt{rds}^v_x,i)}_Q (S,C,P_i[v]) \qquad \text{where } x \notin \mathit{dom}(C_i) \wedge S(x) = v$$

$$(\mathit{ReadPull}) \quad (S,C,P_i[x]) \xrightarrow{(\mathtt{rdp}^v_x,i)}_Q (S,C_i[x \leftarrow (v,\mathtt{clean})],P_i[v]) \quad \text{where } x \notin \mathit{dom}(C_i) \wedge S(x) = v$$

$$(\mathit{WriteBack}) \quad (S,C,P_i[x{:=}v]) \xrightarrow{(\mathtt{wr}^v_{x[i]},i)}_Q (S,C_i[x \leftarrow (v,\mathtt{dirty})],P_i[()])$$

Figure 2: Implementation semantics for read and write operations on a clustering Q

$$(\mathit{Evict}) \quad (S,C,P) \xrightarrow[i\uparrow]{x}_Q (S,C_i \uparrow x,P) \qquad C_i[x] = (v,\mathtt{clean})$$

$$(\mathit{CacheUpd}) \quad (S,C,P) \xrightarrow[i \to j]{x}_Q (S,C_j[x \leftarrow (v,\mathtt{clean})],P)$$
$$x \in \mathit{dom}(C_j) \wedge C_j[x] \neq (v,\mathtt{clean}) \wedge C_i[x] = (v,\mathtt{dirty})$$

$$(\mathit{StoreUpd}) \quad (S,C,P) \xrightarrow[i \to S]{x}_Q (S[x \leftarrow v],C_i[x \leftarrow (v,\mathtt{clean})],P)$$
$$\forall j : j \neq i \wedge x \in \mathit{dom}(C_j), C_j[x] = (v,\mathtt{clean}) \wedge C_i[x] = (v,\mathtt{dirty})$$

Figure 3: System transitions on clustering Q: Update-based cache consistency protocol

If $x \in \mathit{dom}(C_i)$, its value $C_i[x]$ is given by a pair $(\mathit{val},\mathit{state})$, where *val* is the value of the variable and *state* may be either `clean` or `dirty`. A variable is `clean` either if it has not been written to by *this* cluster of cores, or if its changed value has been written through to the store. Otherwise it is `dirty`. Note that in general, $C_i[x] = (v,\mathtt{clean}) \not\Rightarrow S(x) = v$. The system may allow the store to contain a different value if some other processor has updated the store but this cache has not yet been notified.

Figure 2 gives the implementation semantics *with respect to clustering* Q for read and write operations, both of which access the memory — and potentially alter it. When a variable is written to, the write is only to the cache ("write back"). We discuss below how these changes are propagated to the store or to other caches. Observationally, $\mathtt{wr}^v_{x[i]}$ is a *functionally equivalent* action to $\mathtt{wr}^v_x$, but with lower cost: $c(\mathtt{wr}^v_{x[i]}) \simeq \kappa < \delta$.

There are three transitions for reading a variable, $\mathtt{rdl}^v_x, \mathtt{rds}^v_x, \mathtt{rdp}^v_x$, all functionally equivalent to the same specification operation $\mathtt{rd}^v_x$, but with different costs. $\mathtt{rdl}^v_x$ is a read from the local cache, and has cost $c(\mathtt{rdl}^v_x) \simeq \kappa$. Note that when $x \notin \mathit{dom}(C_i)$, there are two possible transitions, labelled $\mathtt{rds}^v_x$ and $\mathtt{rdp}^v_x$, both with costs δ, corresponding to whether or not x is pulled into the cache. This decision is made non-deterministically, which (along with another transition for *eviction* to be introduced later) makes the model independent of the *cache-replacement* policy used by the actual implementation. Note that unlike in the specification semantics, *reading* a location x can cause changes to the memory, e.g., by moving the value read into a cache.

Apart from the *programmed transitions* of Figures 1 and 2, there are the so-called 'system' transitions, denoted by $\rightarrow_Q$, used to manage the memory structure, including cache replacement policies and consistency. These transitions can fire non-deterministically at any time, and the threads cannot constrain which system transitions can occur or when. The system transitions are used to propagate writes to other caches and the store. In practice this is usually done either with an *update-based* protocol (where cached copies are updated with the new value) or with an *invalidation-based* protocol (where cached copies are invalidated, effectively removing them from the cache). Here we present only the update protocol (Figure 3). The transitions, which are not observable, but decorated here to distinguish them, are as follows:

1. **Eviction $\xrightarrow[i\uparrow]{x}$:** Evict x from C_i. $c(\xrightarrow[i\uparrow]{x}) \simeq 0$. This is only used for the cache replacement policy and

is not needed to achieve a consistent state.

2. **Cache update $\xrightarrow[i\to j]{x}$:** Update x in C_j from C_i. $c(\xrightarrow[i\to j]{x}) \simeq 0$ if $i \sim_Q j$, since C_i is the same as C_j; otherwise it is $\simeq \delta$ since it requires communication over the system bus. This is used to update other caches when a variable is written to in a cache.
3. **Store update $\xrightarrow[i\to S]{x}$:** Update x in S from C_i. $c(\xrightarrow[i\to S]{x}) \simeq \delta$. The condition for its application ensures that a store update only happens *after* all caches have been updated and agree on the value of the variable.

3.2 Comparing the semantics on different configurations

Consider a program or workload W, i.e., a set of threads mapped to a set of cores. An execution trace σ of an implementation of W on clustering Q is considered correct if for any two actions $a, b \in \sigma$, some functionally equivalent actions a', b' both appear in the pomset semantics, and if a' precedes b' in the pomset semantics, a appears before b in σ. The implementation conforms to the pomset semantics if every trace σ possible in that implementation is correct with respect to the pomset semantics. In the interleaving view, this can be stated as: for any observable trace in the reference semantics, there is a corresponding trace of functionally equivalent observable actions in the implementation semantics.

Running any workload W on a coarser clustering preserves the observable behaviour. Moreover, CMP is semantically faithful to the reference semantics.

Proposition 1 *1. If $Q' \leq Q$, then any Q-trace σ has a functionally equivalent Q'-trace σ'.*

2. Every reference semantics trace σ has a functionally equivalent CMP-trace σ' and vice versa.

Proof outline: By induction on σ, we find a functionally equivalent σ'. The only interesting cases are if $i \sim_Q j$ but $i \not\sim_{Q'} j$, and there is a local read to or write from (say) $C_i[x]$. We use the $\xrightarrow[i\to j]{x}_{Q'}$ transition before a read or after a write to make the two cache entries agree. Similarly, the $\xrightarrow[i\to S]{x}$ transition is used to make $C_i[x]$ agree with $S[x]$. □

Coherence, Consistency and Data Race Free programs. Unfortunately, program execution on multiprocessors with caches may exhibit more traces than the reference model allows, due to the introduction of race conditions and inconsistencies between the caches at different cores or with the shared store (arising from the non-atomicity of writes). It is therefore not true in general that program behaviour is preserved when running a program on a finer clustering (e.g., SMP).

We recall below some of our earlier results showing that for a class of programs that are "*data race free* (DRF)" [1], every program trace in any implementation architecture has a functionally equivalent trace in the reference architecture. For such DRF programs, the additional behaviours, introduced by the extra nondeterminism in the implementation architecture, are irrelevant for executions starting from "*consistent states*". A consistent state is, intuitively, an implementation state that is identifiable in a precise sense with a specific state (called its "reduct") in the reference model.

We briefly recall the notions of coherence and consistency presented in our earlier work [2] via abstract operational characterisations. We refer the interested reader to *op. cit.* to check that these notions correspond with more familiar invariants associated with memory consistency and cache coherence presented in the literature.

We write $\mathbb{C} \to_Q^* \mathbb{C}'$ to denote that $\mathbb{C}'$ is reachable from $\mathbb{C}$ by the *implementation semantics* (program and system transitions) with respect to clustering Q, and similarly $\mathbb{C} \to_S^* \mathbb{C}''$ for the reference architecture

semantics. We use $\stackrel{*}{\rightarrow}_{\langle Q\rangle}$ to denote 0 or more *system* transitions in clustering Q, whereas $\rightarrow^*_Q$ means 0 or more system and program transitions.
Let us call a state $\mathbb{C}$ "$\rightarrow_{\langle Q\rangle}$-normal" if it cannot make any $\rightarrow_{\langle Q\rangle}$ moves (i.e. system transitions). For an implementation state $\mathbb{C}$, let $\mathbb{C}.M_i[x].val$ denote the value in core i's view of x, i.e., the value of $C_i[x]$, or $S[x]$ if $x \notin dom(C_i)$. An implementation state $\mathbb{C}$ is said to *reduce to* a specification state $\mathbb{C}_S$ (written $\mathbb{C} \Downarrow_Q \mathbb{C}_S$) if $\exists \mathbb{C}' : \mathbb{C} \stackrel{*}{\rightarrow}_{\langle Q\rangle} \mathbb{C}'$, $\mathbb{C}'$ is $\rightarrow_{\langle Q\rangle}$-normal, and $\forall i \forall x\, \mathbb{C}'.M_i[x].val = \mathbb{C}_S.S[x]$. $\mathbb{C}_S$ is called a *reduct* of $\mathbb{C}$.

Definition 1 *A state $\mathbb{C}$ is said to be* coherent for x *if* $\exists v : \forall i : x \in dom(C_i) \wedge C_i[x].state = \texttt{dirty} \Rightarrow C_i[x].val = v$. *A state is* coherent *if it is coherent for all x.* A coherent state has a unique reduct. *We use $\ulcorner\mathbb{C}\urcorner$ to refer to the unique reduct of a coherent state $\mathbb{C}$.*

Definition 2 *A state (S,C,P) is said to be* consistent for x *if and only if* $\forall i : x \in dom(C_i), C_i[x] = (S(x), \texttt{clean})$. *Implementation state $\mathbb{C}$ is* consistent *if it is consistent for all x. A consistent state is in some sense identifiable with its reduct.*

We now introduce the notion of data race freedom.

Definition 3 *A consistent state $\mathbb{C}$ involves a* data race *if it has two redexes $P_i[r]$ and $P_j[r']$, $i \neq j$, r and r' are both accesses to the same variable and at least one is a write. $\mathbb{C}$ is* data race free *(DRF) iff no state reachable in the reference architecture semantics from $\ulcorner\mathbb{C}\urcorner$ involves a data race.*

Note also that the analysis of data race freedom need only be performed at the level of the reference semantics. DRF programs allow us to consider their execution as progressing via a sequence of reference model states (reducts). Using ideas and techniques introduced by Boudol and Petri [1], it is shown that for "well-synchronised" programs, i.e., those where between any pair of actions forming a data race lies an intervening synchronising mechanism (e.g., lock release or barrier operations), the behaviours are functionally equivalent. In particular, we have shown in [2, Section 6] that the cache rules described above satisfy required properties of coherence and consistency.

Since the execution of DRF programs on any clustering architecture can be seen to be functionally equivalent to execution on the reference architecture, we can show:

Proposition 2 *For DRF programs, any reconfiguration (in any direction) preserves semantics.*

Proof outline: Similar to the proof of Proposition 1, but here we rely on the fact that every implementation trace of a DRF programs on clustering Q has a functionally equivalent trace in the reference model. The result follows from transitivity of equivalence of actions, and that 'system' transitions are not directly observable. □

Execution on a dynamically reconfiguring architecture. Consider now a scenario where execution may commence on clustering Q, and the machine may nondeterministically decide to morph to a clustering Q' (based on some heuristic), after which execution proceeds on the latter architectural configuration. Such *dynamic reconfiguration* from clustering Q to Q' can be internalised into our framework by introducing a new action ${}_Q\!\!\not\rightarrow^{Q'}$, with cost $c({}_Q\!\!\not\rightarrow^{Q'}) = \mu \gg \delta$:

$$(Reconf) \quad \mathbb{C} \;{}_Q\!\!\not\rightarrow^{Q'} \ulcorner\mathbb{C}\urcorner$$

where $\ulcorner\mathbb{C}\urcorner$ is the "reduct" of $\mathbb{C}$. On reconfiguration, the cache contents are written to the store, and the program resumes in the new configuration with "cold caches". For DRF programs, the execution semantics of each phase corresponds precisely to the semantics of execution in the reference model. Thus we can formalise within our framework the correctness criterion for execution on a reconfigurable architecture.

Theorem 1 *Any execution of a DRF program on a reconfigurable architecture conforms to execution on the reference model.*

Proof outline: By piecewise stitching of executions of the different phases on the different configurations. Note that implementation states before and after reconfiguration correspond to the same reduct. □

4 Comparing Performance

We now propose a framework for comparing the execution efficiency on two configurations. For uniformity, specification and implementation states are clubbed into one set; the specification and implementation actions marking transitions are also combined into one set of actions $\mathscr{A}$. Let $u \in \mathscr{A}^*$ be a sequence of actions, and let $\stackrel{u}{\Longrightarrow}$ denote a labelled sequence of transitions, as usual.

Let $\rho \subset \mathscr{A} \times \mathscr{A}$ represent the functional equivalence of actions as mentioned above. The actions $\mathtt{rdl}^v_x, \mathtt{rds}^v_x, \mathtt{rdp}^v_x, \mathtt{rd}^v_x$ are in one equivalence class; $\mathtt{wr}^v_{x[i]}$ and $\mathtt{wr}^v_x$ in a second; and $\xrightarrow[i\uparrow]{x}, \xrightarrow[i\to j]{x}, \xrightarrow[i\to S]{x}, \tau, \not\mapsto$ in the third. For α in the read and write actions, let the observable content $\hat{\alpha} = \alpha$, and for $\alpha \in \{\xrightarrow[i\uparrow]{x}, \xrightarrow[i\to j]{x}, \xrightarrow[i\to S]{x}, \tau, \not\mapsto\}$, define $\hat{\alpha} = \varepsilon$ (the empty string). Extend ρ to sequences such that $(\varepsilon,\varepsilon) \in \rho$ and $(\varepsilon, u_1...u_n) \in \rho$ and $(u_1...u_n, \varepsilon) \in \rho$ if $(\tau, u_i) \in \rho$ for each u_i.

Functionally equivalent actions deliver the same results but may have quite different latencies. We have earlier specified the costs of actions κ for cache accesses, δ for store accesses and μ for reconfiguration. Lift $c(\ )$ to sequences by summing the costs of the component actions.

Following the constraints on ρ and latency costs as in [3], we define the notion of weak amortised bisimulations on states (both specification and implementation)[3].

Definition 4 *A family $(R_i)_{i\in\mathbb{N}}$ of binary relations over states is a* weak amortised ρ-bisimulation*, if for all $i \in \mathbb{N}$ whenever $(\mathbb{C}_1, \mathbb{C}_2) \in R_i$:*

$\mathbb{C}_1 \xrightarrow{a} \mathbb{C}'_1$ implies $\exists \mathbb{C}'_2, b, u, v :\ (a,b) \in \rho, (\varepsilon,u) \in \rho, (\varepsilon,v) \in \rho,\ \mathbb{C}_2 \stackrel{u\hat{b}v}{\Longrightarrow} \mathbb{C}'_2$ and $(\mathbb{C}'_1, \mathbb{C}'_2) \in R_{i+c(u\hat{b}v)-c(a)}$,

$\mathbb{C}_2 \xrightarrow{b} \mathbb{C}'_2$ implies $\exists \mathbb{C}'_1, a, u, v :\ (a,b) \in \rho, (u,\varepsilon) \in \rho, (v,\varepsilon) \in \rho,\ \mathbb{C}_1 \stackrel{u\hat{a}v}{\Longrightarrow} \mathbb{C}'_1$ and $(\mathbb{C}'_1, \mathbb{C}'_2) \in R_{i+c(b)-c(u\hat{a}v)}$,

where $a, b \in \mathscr{A}$ and $u, v \in \mathscr{A}^*$. State $\mathbb{C}_1$ is (weakly) amortised more efficient than $\mathbb{C}_2$ up to credit i, written $\mathbb{C}_1 \preceq^{\rho}_i \mathbb{C}_2$, if $(\mathbb{C}_1, \mathbb{C}_2) \in R_i$ for some weak amortised ρ-bisimulation $(R_i)_{i\in\mathbb{N}}$.

Note that accessing the cache is significantly less costly that accessing the store. The definition of weak amortised bisimulation accumulates "credit" by performing the cheaper operation, thus providing us a framework for comparing the performance of execution on different architectural configurations. Since there is nondeterminism in when the 'system' operations take place, our framework must account for every possible execution run in the comparison.

The notion also allows us to assess the benefits of performing dynamic reconfiguration. Reconfiguration introduces a handicap of μ, which must be balanced (in an amortised sense) by frequent accesses to cache instead of to the store. Typically δ is about 4 instruction cycles whereas κ is one cycle. Since values are pulled into cache in blocks of words, there are additional performance benefits due to overlapping reads with the execution of other operations. The reconfiguration cost μ is approximately 1000 instruction cycles, so if there is enough locality of reference, this reconfiguration cost may be easily offset by the benefits of running the workload on a more suitable configuration for phases that are typically of the order of millions of instruction cycles or more.

[3] A major difference with the cited work is that we ascribe a cost to observable actions as well

Theorem 2 *1. For DRF program states, any reconfiguration for more efficiency is permissible. With sufficient locality of references, such programs executed on any clustering are "weakly amortised" more efficient than execution under their reference model execution.*

2. If $Q \leq Q'$, then executing a program on configuration Q' is (modulo the approximations on latency) "weakly amortised more efficient" than on Q.

Proof outline: From any consistent implementation state, a unique reduct is reachable. If it takes $m \leq kn$ $\xrightarrow[i\to j]{x}, \xrightarrow[i\to S]{x}$ moves to do so (where k is the number of caches and n the number of variables), then to have been in such a state, the system must have earlier performed at least m $\mathtt{wr}^v_{x[i']}$ actions instead of $\mathtt{wr}^v_x$ actions, and thus have already "earned" a credit of $m(\delta-\kappa)$. So if it has performed at least $m\kappa/(\delta-\kappa)$ operations such as distinct repeat writes to a variable in cache or reads of a `dirty` variable, then it has earned the requisite credit to be amortised at least as efficient as the reference system.
For the second part, if $Q \leq Q'$, since there is more sharing of cached variables, we can avoid the costlier $\mathtt{rds}^v_x, \mathtt{rdp}^v_x$ in favour of the cheaper $\mathtt{rdl}^v_x$ operation, and avoid some instances of the $\xrightarrow[i\to j]{x}$ operation. □

5 Conclusions

Our framework provides a formal basis for comparing both the behaviour and the performance of program workloads on different multiprocessor architectures. Our first set of results (Propositions 1 and 2) provides us a rudimentary formal justification for the folklore that it is easy to port programs written assuming an SMP architecture to CMP. In particular, they indicate why converting MPI programs to OpenMP, which assumes shared variables, is usually easier than the trickier reverse direction. Theorem 1 indicates why as architectures become "smarter" and incorporate reconfigurability, avoiding data races only assumes greater importance. The framework for comparing the efficiency of execution on different architecture also lets us understand why certain programs get such dramatic performance benefits when run on CMP architectures.

Let us mention a shortcoming of our work. Theorem 2 seems to indicate that CMP is preferable to all other configurations, which is belied in reality, especially where applications with threads having little or no communication amongst themselves run more efficiently on SMP-like configurations. The cache bus within a cluster and the memory bus enforce mutually exclusive access, and so the interleaved execution of threads accessing a shared cache is slower than simultaneous independent execution of threads accessing private caches ($\kappa+\kappa$ interleaved vs κ in parallel)[4]. The fault lies not in our framework, but rather in our view of execution as being interleaved — a view we had taken to keep the semantics standard. We leave for the future the development of a framework where the pomset model is used to explore the correctness and efficiency issues.

Reconfiguration is also an opportunity for remapping threads to cores. While we have considered only a fixed workload mapping of threads to cores in the present paper, we do not believe this extension poses any major technical difficulties.

Acknowledgements. I wish to acknowledge the helpful discussions with my colleague Kolin Paul who taught me the little I know about reconfigurable architectures. I also must thank one of the referees who constructively pointed out many major weaknesses of the earlier avatar of this paper.

[4]There also is a slowdown due to the fused cache being larger, which we neglect.

References

[1] Gérard Boudol & Gustavo Petri (2009): *Relaxed memory models: an operational approach*. In: *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pp. 392–403, doi:10.1145/1480881.1480930.

[2] Salil Joshi & Sanjiva Prasad (2010): *An Operational Model for Multiprocessors with Caches*. In: *Theoretical Computer Science - 6th IFIP TC 1/WG 2.2 International Conference, TCS 2010, Held as Part of WCC 2010, Brisbane, Australia, September 20-23, 2010. Proceedings*, pp. 371–385, doi:10.1007/978-3-642-15240-5.

[3] Astrid Kiehn & S. Arun-Kumar (2005): *Amortised Bisimulations*. In: *Formal Techniques for Networked and Distributed Systems - FORTE 2005, 25th IFIP WG 6.1 International Conference, Taipei, Taiwan, October 2-5, 2005, Proceedings*, pp. 320–334, doi:10.1007/11562436_24.

[4] Padmanabhan Krishnan (1992): *A Semantics for Multiprocessor Systems*. In: *ESOP '92, 4th European Symposium on Programming, Rennes, France, February 26-28, 1992, Proceedings*, pp. 307–320, doi:10.1007/3-540-55253-7_18.

[5] Padmanabhan Krishnan (1996): *Architectural CCS*. *Formal Asp. Comput.* 8(2), pp. 162–187, doi:10.1007/BF01214555.

[6] L. Lamport (1979): *How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs*. *IEEE Trans. Comput.* 28(9), pp. 690–691, doi:10.1109/TC.1979.1675439.

[7] Rajesh Kumar Pal, Kolin Paul & Sanjiva Prasad (2012): *ReKonf: A Reconfigurable Adaptive ManyCore Architecture*. In: *10th IEEE International Symposium on Parallel and Distributed Processing with Applications, ISPA 2012, Leganes, Madrid, Spain, July 10-13, 2012*, pp. 182–191, doi:10.1109/ISPA.2012.32.

[8] Rajesh Kumar Pal, Kolin Paul & Sanjiva Prasad (2014): *ReKonf: Dynamically reconfigurable multiCore architecture*. *J. Parallel Distrib. Comput.* 74(11), pp. 3071–3086, doi:10.1016/j.jpdc.2014.05.007.

[9] Vaughan R. Pratt (1984): *The Pomset Model of Parallel Processes: Unifying the Temporal and the Spatial*. In: *Seminar on Concurrency, Carnegie-Mellon University, Pittsburg, PA, USA, July 9-11, 1984*, pp. 180–196, doi:10.1007/3-540-15670-4_9.