Petri-nets as an Intermediate Representation for Heterogeneous Architectures

Peter Calvert and Alan Mycroft

Computer Laboratory, University of Cambridge
William Gates Building, JJ Thomson Avenue,
Cambridge CB3 0FD, UK
firstname.lastname@cl.cam.ac.uk

Abstract. Many modern systems provide heterogeneous parallelism, for example NUMA multi-core processors and CPU-GPU combinations. Placement, scheduling and indeed algorithm choices affect the overall execution time and, for portable programs, must adapt to the target machine at either load-time or run-time. We see these choices as preserving I/O determinism but exposing *performance non-determinism*. We use Petri-nets as an intermediate representation for programs to give a unified view of all forms of performance non-determinism. This includes some scenarios which other models cannot support. Whilst NP-hard, efficient heuristics for approximating optimum executions in these nets would lead to performant portable execution across arbitrary heterogeneous architectures.

Keywords: Petri-nets, parallelism, heterogeneous architectures, scheduling.

1 Introduction

It is becoming clear that modern systems are not only increasingly parallel but also heterogeneous. Common examples include IBM's Cell Broadband Engine and also CPU-GPU combinations. As well as different processing capabilities, the different cores have access to separate memories, with data transfers needing to be managed explicitly. Our research is focused on offering portability for these systems.

On such architectures, achieving optimal performance depends on careful placement of computation and management of the required data transfers. If we want programs to have *portable* performance, this placement must be done automatically. This is an example of *performance non-determinism*, where run-time or load-time decisions affect the overall execution time. A choice between algorithms also causes this (e.g. the fastest sequential algorithm compared to one well suited to parallelisation). We distinguish this from *I/O non-determinism*, where run-time decisions might alter the *result* of the program.

Other work on heterogeneous architectures tends to treat every task as a single unit that must be completed, just giving it a different cost for each processor

E. Jeannot, R. Namyst, and J. Roman (Eds.): Euro-Par 2011, LNCS 6853, Part II, pp. 226–237, 2011.
© Springer-Verlag Berlin Heidelberg 2011

[14]. However, this does not allow scenarios such as *either perform A once at cost 5 or perform B ten times (possibly in parallel) at cost 1 each*. We wish to encode these possibilities.

In this paper, we use *coloured Petri-nets* as an intermediate representation for parallel programs, and investigate the placement and scheduling problems. In particular, we consider:

- A Petri-net intermediate representation that expresses both parallelism and performance non-determinism, and some example programs (Section 3).
- A simple model of heterogeneous architectures and how this provides a unified model of existing hardware (Section 4).
- How these Petri-nets map onto such hardware, including a cost model and the problem of minimising execution time (Section 5).
- Existing work from both mathematics and computer science that addresses issues that we identify in this minimisation problem (Section 6).
- How compiler optimisations can be applied to these Petri-nets (Section 7).

We also discuss other parallel models and their relation to this work.

2 Notation

Throughout this paper, we make frequent use of *sets*, *multisets* and *ordered lists*. The notation that we will use is as follows:

- $\mathbb{N}_0$ gives the set of non-negative integers—i.e. $\{0, 1, 2, \dots\}$.
- X^∞ gives the set $X \cup \{\infty\}$.
- $\mathbb{R}_+$ gives the set of non-negative real numbers.
- A multiset m over a set X is a function $X \to \mathbb{N}_0$ giving the number of appearances $m(x)$ of $x \in X$ in m. The set of all multisets is written $\mathbf{m}X$. The operations $\cup$, $\setminus$ and $\subseteq$ are defined as liftings of $+$, saturating subtraction and $\leq$ respectively.
- A list l over a set X is a tuple X^n. We will write l_i for the ith element of l (for $i = 1 \dots n$) and $|l|$ for its length n. The set of all lists is written $\ell X = \bigcup_{n \in \mathbb{N}_0} X^n$. Note that we can treat lists as multisets, but not vice-versa.

We also use a very simple type system:

$$\tau ::= \text{int} \,|\, \text{bool} \,|\, \text{unit} \,|\, \tau \times \tau \,|\, \tau[n] \quad \text{for } n \in \mathbb{N}$$

It is important that each type is of fixed size, given by $\text{sizeof}(\tau)$. Therefore, arrays of different lengths are distinct types. The set of values that a type τ can take is denoted $[\![\tau]\!]$. We do not have higher order types but will refer to transitions of type $\tau \to \tau'$, and write for functions:

$$f : \tau \to \tau' \iff \forall v \in [\![\tau]\!] : (f(v) \text{ defined} \implies f(v) \in [\![\tau']\!])$$

Later, we will abuse $[\![_]\!]$ to give the operation associated with a transition. When referring to the cost of operations, we will generally use the notation $\langle _ \rangle$.

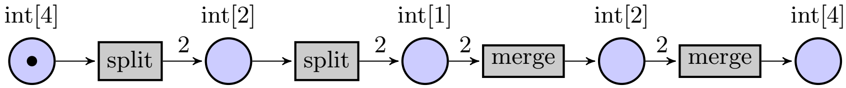


Fig. 1. Petri-net for *merge-sort* of a 4-element list

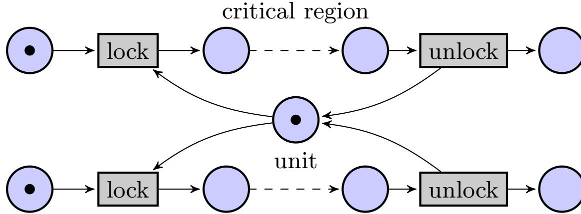


Fig. 2. Petri-net for a critical region using a *mutual exclusion lock*

3 Petri-net Intermediate Representation

Any intermediate representation for modern architectures must express parallelism, and, we argue, performance non-determinism. The current common formats are *data-flow graphs (DFGs)* and *control-flow graphs (CFGs)*.

Data-flow graphs express parallelism, but not non-determinism. Every computation node in the graph will be performed at some point, and the edges give the dependencies between them. On the other hand, control-flow graphs express no parallelism at all, but branching is often treated as non-determinism (e.g. in static analysis).

Petri-nets are the obvious combination of these two, offering both parallelism and non-determinism. We consider a variant of Jensen's *coloured Petri-nets* or *CP-nets* [8]. *Tokens* (drawn •) containing values are stored at *places* (drawn ◯) which have a type. Execution proceeds by the firing of *transitions* (drawn □). When a transition *fires*, it removes tokens from its *pre-places*, applies a function to them, and adds the results to its *post-places*. Later, we will allow such transitions to take a period of time. For example, a simple merge-sort of a 4-element list can be represented as shown in Figure 1 (the '2's in the diagram show multiplicities where multiple tokens are taken from, or given to, a single place in a firing). Petri-nets also express concurrency primitives well, for example mutual exclusion (Figure 2).

In our nets, the effect of a transition is defined using an existing representation for sequential programs (e.g. functions in LLVM's IR [1] or even individual virtual machine instructions). However, for this work, the exact choice is not important, we simply refer to the set $\mathbb{F}$ of such partial functions[1].

[1] We elide the difference between intensional and extensional representations of a function, so for $f_1, f_2 \in \mathbb{F} : (\forall x : f_1(x) = f_2(x)) \not\Rightarrow f_1 = f_2$ since f_1 and f_2 may compute the result in different ways.

The partiality of these functions provides for conditional transitions. If a transition is not defined for a given input, it will not fire. For example, the following transition *cond* of type $bool \rightarrow unit$ will generate a token (with *unit* value) only when provided with a token containing the value **true**:

$$[\![\text{cond}]\!](b) = \begin{cases} () & \text{if } b = \text{true} \\ \text{undefined} & \text{if } b = \text{false} \end{cases}$$

We can now define our version of coloured Petri-nets more formally:

Definition 1. *A* **coloured Petri-net** *is a tuple* $N = (S, T, \Gamma, {}^\bullet_, _^\bullet, [\![_]\!])$ *consisting of:*

1. *A set of* places S.
2. *A set of* transitions T.
3. *A* type environment Γ, *associating a type* τ *to every* $s \in S$.
4. *A* pre-place *function* ${}^\bullet_ : T \rightarrow \ell S$.
5. *A* post-place *function* $_^\bullet : T \rightarrow \ell S$.
6. *A* labelling *function* $[\![_]\!] : T \rightarrow \mathbb{F}$ *such that transitions are well typed—i.e.*

$$\textit{For all } t \in T,\ [\![t]\!] : \Gamma({}^\bullet t) \rightarrow \Gamma(t^\bullet)$$

where $\Gamma([s_1, \ldots, s_n]) = \Gamma(s_1) \times \cdots \times \Gamma(s_n)$.

Note that we have defined the pre- and post-places of transitions as *lists* rather than *multisets* to give direct association with argument and result tuples of functions in $\mathbb{F}$.

The state of a coloured Petri-net describes the tokens, and their values, present at each place. This is called a *marking* (although Jensen [8] prefers *token distribution* when tokens carry values) and can be defined as follows:

Definition 2. *A* **marking** *is a function* M *defined on* S *such that* $M(s) \in \mathbf{m}[\![\Gamma(s)]\!]$ *for all* $s \in S$. *We denote the set of all markings as* $\mathbb{M}$. *The operators* $\cup$, $\setminus$ *and* $\subseteq$ *lift to markings in the obvious manner.*

Given a list of places $i \in \ell S$ (e.g. the pre-places or post-places of a transition), and a list of values $\mathbf{x} \in [\![\Gamma(i)]\!]$ (i.e. which are well typed), the corresponding marking[2] will be written $(i \rightsquigarrow \mathbf{x})$.

This allows us to define the *firing rule* $M \rightarrow M'$ of the Petri-net as follows:

$$M \cup ({}^\bullet t \rightsquigarrow \mathbf{x}) \rightarrow M \cup (t^\bullet \rightsquigarrow [\![t]\!](\mathbf{x})) \text{ provided that } [\![t]\!](\mathbf{x}) \text{ is defined}$$

Traditionally, paths through Petri-nets are represented either as *causal nets* [12] or by *pomsets* [10]. When we come to introducing a cost model in Section 5, it will be more convenient to use pomsets.

[2] For example, for $A, B, C \in S$ and $\Gamma(s) = \text{int}$ for $s \in \{A, B, C\}$, $([A, B, C, C] \rightsquigarrow [10, 4, 5, 2]) = (A \mapsto \{10\}; B \mapsto \{4\}; C \mapsto \{5, 2\})$.

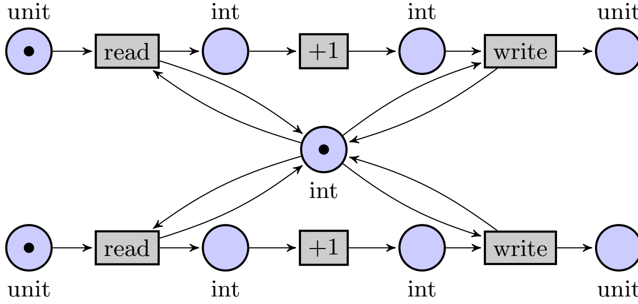


Fig. 3. Petri-net without confluence ($x := x + 1 \parallel x := x + 1$)

Definition 3. *A* **(pomset) path** *is a triple* $(V, \leq, \mu)$ *where* V *is a set of occurences labelled with a transition by* $\mu : V \to T$*.* $\leq$ *defines a partial order on* V*. Two pomset paths are considered equivalent if there is an isomorphism between them.*

We will use $\mathbb{P}$ to give the set of all such paths. Note that for two transiton occurences $v_1, v_2 \in V$, if neither $v_1 \leq v_2$ nor $v_2 \leq v_1$ then the occurences can occur in parallel, whereas otherwise they must fire sequentially.

Just as with traditional flow graphs, not all of these paths are *feasible*, since we cannot check whether the transitions are defined without knowing the values of the tokens. However, any execution trace will be an instance of a path from a distinguished initial place s_0 to a final place s_∞. It must not be possible to fire any transitions from the final place (i.e. $\forall t \in T : s_\infty \notin {}^\bullet t$).

The firing of Petri-nets is non-deterministic, as intended. However, this allows all forms of non-determinism to be expressed, not just performance non-determinism (e.g. Figure 3). We will assume that programs are I/O deterministic, and therefore respect *confluence*—i.e. for all markings $M_1, M_2, M_3 \in \mathbb{M}$:

$$(M_1 \to^* M_2) \wedge (M_1 \to^* M_3) \implies \exists M_4 \in \mathbb{M}.((M_2 \to^* M_4) \wedge (M_3 \to^* M_4))$$

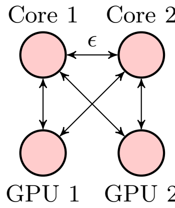
4 Simple Hardware Model

We restrict ourselves to a very simple model of heterogeneous architectures. This ignores fine details of the memory system such as caches. We consider a system to consist of *processors*, each with a *local memory*, and *interconnects* between them. The cost of accessing this local memory is low and included in the computation cost of a function. Non-local data must be transferred via interconnects before use, at a cost modelled by latency and bandwidth. This is not dissimilar from the partitioned global address space model (PGAS) that is used elsewhere (e.g. X10). We ignore capacity constraints of memories. Formally, a hardware architecture is defined as follows:

Definition 4. *A* **simple heterogeneous hardware model** H *is a 3-tuple* (P, m, c) *consisting of:*

- *A finite set of* processors P.
- *An* interconnect descriptor *function* $i : (P \times P) \rightarrow (\mathbb{R}_+ \times \mathbb{R}_+)^\infty$. *For a pair* (p_1, p_2) *of distinct processors,* $i(p_1, p_2) = (l, b)$ *gives the latency* l *and per-byte cost* b $(= \frac{1}{bandwidth})$ *of the interconnect from* p_1 *to* p_2*. We will refer to the cost of transferring* n *bytes of data with the notation* $\langle p_1 \xrightarrow{n} p_2 \rangle = l + n \cdot b$. *When there is no interconnect from* p_1 *to* p_2*,* $i(p_1, p_2) = \infty$.
- *A* computation cost *function* $c : (\mathbb{F} \times P) \rightarrow \mathbb{R}_+^\infty$*, where* ∞ *indicates that the processor cannot perform the function (e.g. no floating-point support).*

An example model of a multi-core plus GPU architecture is given in Figure 4. The inclusion of small costs, such as ϵ in the example, approximates the effect of cache invalidations, when cores share a memory but have separate caches. Memories not associated with a processor can be modelled as a 'null' processor $p_\perp$ with $c(f, p_\perp) = \infty$ for all $f \in \mathbb{F}$.



Here ϵ is typically small, and the other costs are based on actual measurements.

Fig. 4. Example model for dual-core CPU with 2 general-purpose GPUs

We assume two sanity constraints: that the memory interconnect is strongly connected[3], and also that all functions can be executed somewhere (i.e. $\forall f \in \mathbb{F} : \exists p \in P : c(f, p) \neq \infty$). These properties ensure that our mapping of software onto hardware is also confluent.

5 Mapping Software to Hardware

Given these two models, we can model all possible executions of a program on an architecture with a single Petri-net. Each feasible path through the net gives a possible execution trace. The intuition behind our construction comes from considering an individual data token x. In program $N = (S, T, \Gamma, {}^\bullet_, _^\bullet, [\![_]\!])$, x must be at some place $s \in S$. However, the architecture $H = (P, m, c)$ on which the software is run, must store x in some memory $p \in P$. Therefore, the location of a *data* token in a running program is described by a pair from the set $S \times P$.

We now consider what might happen to a token x at (s, p). There are two options, either:

- (t, p): The token x is used by transition $t \in T$ executing on processor p (where possible), *or*

[3] A graph is strongly connected if for every pair of vertices a and b, there is a path both from a to b, and b to a.

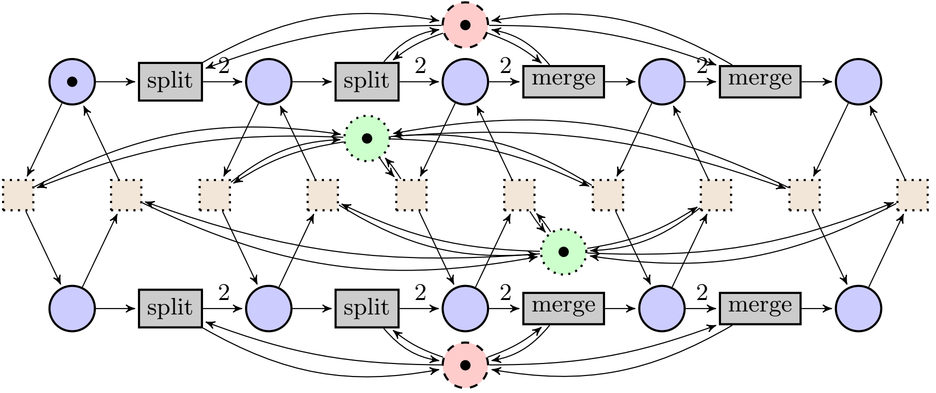


Fig. 5. *Merge-sort* mapped onto a dual core CPU (types are omitted for clarity, and memory transfers are only shown for $n = 1$)

– (s, p, p', n): The token x is transferred to another processor $p' \in P$ via an interconnect (p, p') as part of an n-token transfer.

Therefore, the possible transitions for x must be a subset of $(T \times P) \cup (S \times P^2 \times \mathbb{N})$. Additionally, when we consider the complete system, we require *resource constraints* so that a processor executes only a single transition at any moment, and similarly so that each interconnect is only used for one transfer at a time.

We can encode this as a new Petri-net. To encode the resource constraints, we choose to use mutual exclusion locks similar to Figure 2, although they could also be expressed as restrictions on paths. This results in a net C defined as follows[4]. The Petri-net for our merge-sort example on a dual core CPU is shown in Figure 5 as an example, although we do not intend this model to be a large-scale visual model.

Interconnect Constraints

Data Places

Memory Transfers

$$C = \Big(\big(\underbrace{(S \times P)}_{\text{Data Places}} \cup \underbrace{P}_{\text{Processor Constraints}} \cup \underbrace{P^2}_{\text{Interconnect Constraints}}, \underbrace{(T \times P)}_{\text{Computation Transitions}} \cup \underbrace{(S \times P^2 \times \mathbb{N})}_{\text{Memory Transfers}}, \Gamma', {}^{\bullet}_, _^{\bullet}, [\![_]\!] \Big)$$

Processor Constraints

Computation Transitions

with[5]:

$$\begin{aligned} {}^{\bullet}(t,p) &= [p, ({}^{\bullet}t_1, p), \ldots, ({}^{\bullet}t_{|{}^{\bullet}t|}, p)] & {}^{\bullet}(s, p_1, p_2, n) &= [(p_1, p_2), (s, p_1), \ldots, (s, p_1)] \\ (t,p)^{\bullet} &= [p, (t_1^{\bullet}, p), \ldots, (t_{|t^{\bullet}|}^{\bullet}, p)] & (s, p_1, p_2, n)^{\bullet} &= [(p_1, p_2), \underbrace{(s, p_2), \ldots, (s, p_2)}_{n \text{ repetitions}}] \end{aligned}$$

[4] Excluding the resource constraints, this construction is equivalent to the Cartesian product of hypergraphs, where each Petri-net transition corresponds to a hyperedge.

[5] For these definitions, we use ML-style list syntax (i.e. $[a, b, c]$ is a list, and $1 :: [2, 3, 4] = [1, 2, 3, 4]$).

and:

$$\Gamma'(s') = \begin{cases} \Gamma(s) & \text{if } s' = (s,p) \\ \text{unit} & \text{otherwise (i.e. resource constraint places)} \end{cases}$$

$$[\![(t,p)]\!] = \lambda(r, x_1, \ldots, x_n)\,.\,() :: [\![t]\!](x_1, \ldots, x_n)$$

$$[\![(s, p_1, p_2, n)]\!] = \lambda(r, x_1, \ldots, x_n)\,.\,[(), x_1, \ldots, x_n]$$

In Section 4, we required that our memory be strongly connected. This ensures that data transfers can always be 'undone'. Similarly, since each function in $\mathbb{F}$ can be done on some $p \in P$, we know that the new Petri-net is still confluent. Therefore, the choice of which transition to fire can only affect performance, not correctness.

Since our hardware model gives us costs, we can supplement our 'compiled' Petri-net with a *duration function* $\langle_\rangle$. This gives the time taken for each transition to fire. It can be defined as follows:

$$\langle(t,p)\rangle = c([\![t]\!], p)$$

$$\langle(s, p_1, p_2, n)\rangle = \langle p_1 \overset{n \cdot \mathrm{sizeof}(\Gamma(s))}{\rightarrow} p_2\rangle$$

Now we have durations associated with each transition, it is reasonable to ask how long a path through C will take to execute. Given a pomset path $\mathfrak{p} = (V, \leq, \mu) \in \mathbb{P}$, this is given by $\langle\mathfrak{p}\rangle = \max_{v \in V}(f(v))$ where the finish time $f(v)$ of an occurence is given by:

$$f(v) = \begin{cases} \max_{\{w \in V | w \leq v\}}(f(w)) + \langle\mu(v)\rangle & \text{if } \exists w \in V : w \leq v \\ \langle\mu(v)\rangle & \text{otherwise} \end{cases}$$

Candidate executions of N on H are given by any paths from an initial place to a final place. Unfortunately, as noted in Section 3, not all paths are feasible. In executing the Petri-net, the aim is to choose a trace with *minimum duration.*

6 Finding Optimal Executions

The problem of finding an optimal execution allows us to consider all performance non-determinism choices together. This is not limited to placement and scheduling, but also programming model specific choices, such as *which thread should get access to the lock first* and *how many times should we perform divide and conquer for our algorithm to best match the available parallelism.*

We can consider the problem in two stages. Firstly, we must be able to find traces of minimum duration where no partial transitions (i.e. conditionals) are used, and therefore all paths are trivially feasible. This corresponds to analysis of straight-line flow graphs. Once solved, extending this to the complete problem will require runtime analysis, since input values will affect which paths can occur. Fortunately, as pointed out in Section 5, we cannot make any *wrong* choices (just slow ones). We might therefore aim to pick transitions that appear in a number of low duration paths.

6.1 Complexity

Even the first part is an NP-hard problem. The proof uses a reduction from the *exact cover* problem, which is very similar to the reduction from *3-dimensional matching* for AND/OR network scheduling [6]. These problems are both known to be NP-complete [9].

Theorem 1. *Checking whether a path exists between two markings is NP-hard.*

Proof. Given a set X, and a set $Y \subseteq \wp(X)$ of subsets, an exact cover is a set $Y^* \subseteq Y$ such that for each element of X it appears in exactly one element of Y^*. Determining whether an exact cover exists for a given X and Y is known to be NP-complete [9].

We can encode this in a Petri-net $(S, T, \lambda s.unit, {}^\bullet_, _^\bullet, \lambda t.(\lambda x.[(), \ldots, ()]))$ as follows. For each $x \in X$, we include a place $s_x \in S$. We also introduce a start place **start**. *Now for each subset $y = \{x_1, \ldots, x_n\} \in Y$, we add a transition $t_y \in T$ such that:*

$$
\begin{aligned}
{}^\bullet t_y &= [start] \\
t_y^\bullet &= [start, s_{x_1}, \ldots, s_{x_n}]
\end{aligned}
$$

We can then determine whether an exact matching Y^* exists by checking whether there is a path from $\{start\}$ to $\{start\} \cup X$.

6.2 Similar Problems and Techniques

Scheduling appears to be a similar problem, and despite being NP-complete, effective heuristics do exist for it. This suggests there may also be approximations for our problem. However, there are several key differences between the problems.

In typical scheduling problems, precedence constraints do not allow the concept of a completed task being 'used up' as in Petri-nets, and cycles therefore behave differently. This *dataflow* situation is seen with *streaming models* (e.g. [11]), where the input and output rates of pipeline stages need to be matched. Adaptive runtime approaches have been shown to produce good results [13] for such pipelines.

Flows through graphs also share similarities, with the flow into and out of each node needing to match (our program outputs a single token at s_∞ so tokens cannot build up at intermediate places). We would need to use *hypergraphs*[6] since transitions have multiple pre- and post-places. The *minimum cost flow* problem is most relevant since we know how much data will be input and expected as output. However, this minimises total cost rather than the critical path.

In the standard graph case, there are efficient *cost-scaling* algorithms [7] for this which could be distributed. However, work on hypergraphs [4] appears to be restricted to 'B-hypergraphs' where each edge has only a single head.

Unfortunately, the standard definition of 'hyperflows' fails to describe execution paths (for example, Figure 6). Also a flow would need to be supplemented by an actual schedule for execution.

[6] A (directed) hypergraph is here a digraph where each hyperedge e has multiple heads and multiple tails (i.e. $e \in \wp V \times \wp V$).

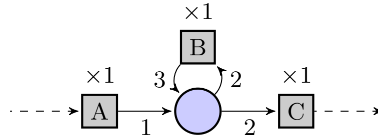


Fig. 6. A hypergraph flow that is not a valid path (A must fire *twice* for B or C to fire)

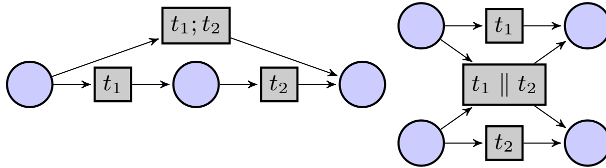


Fig. 7. Fusing of Petri-net transitions

The non-deterministic choice present in Petri-nets is uncommon in scheduling. *AND/OR networks* do share this characteristic. They are typically solved using list scheduling heuristics [5]. However, it is unclear how this can be combined with the dataflow properties defined above.

Graph flows are again similar since a flow will not necessarily use every edge.

7 Compiler Optimisations

Using the representation that we have described may make parallelism explicit, however it does not consider compiler optimisations apart from within a single transition. The potentially fine-grained nature of transitions means that there will be a limited performance improvement. If we enlarge the transitions in the original program, then some of the explicit parallelism may be lost. The solution to this is to allow transitions to be *fused* at load-time or runtime (Figure 7). For two transitions t_1 and t_2 on a *single processor* p, we can hope that[7] $c([\![t_1; t_2]\!], p) < c([\![t_1]\!], p) + c([\![t_2]\!], p)$ or $c([\![t_1 \parallel t_2]\!], p) < c([\![t_1]\!], p) + c([\![t_2]\!], p)$. For example, with SIMD execution, it may be that $c([\![t \parallel t]\!], p) = c([\![t]\!], p)$.

A fusing can be defined by a path in the original program Petri-net, since a path encodes the ways in which transitions can be combined, including repetitions of a single transition. Any given path is also finite, so we avoid the problem of defining a cost for looping code. We can incorporate this into our mapping by creating a transition for each path $\mathfrak{p} \in \mathbb{P}$ in the program rather than for each transition $t \in T$—i.e. the set of transitions in the mapped Petri-net becomes $((\mathbb{P} \times P) \cup (S \times P^2 \times \mathbb{N}))$.

In practice, we cannot enumerate all paths (an infinite set), so heuristics for specific core types would be used. For example, data parallel devices (e.g. GPUs) would be most effective on paths that compose a calculation in parallel with

[7] $t_1; t_2$ and $t_1 \parallel t_2$ give sequential and parallel compositions of t_1 and t_2 respectively.

itself. However, even with heuristics we will generate many transitions. Again looking at data parallel cores as an example, a device that can execute between 1 and N threads will generate a transition for each choice of $n \in \{1, \ldots, N\}$. In the future, we will therefore need to look at how transitions can be parameterised.

Similarly, our current representation does not support any notion of procedures. This becomes impractical for large programs, and we will need to investigate ways of including this. Jensen's hierarchical CP-nets offer one approach [8], but we will also consider techniques from other areas such as hardware description languages (since places behave somewhat like wires).

8 Comparison with Other Models

The best-known bridging model for parallel computation is Valiant's *BSP* [15]. However, this only considers *homogeneous* scenarios. In BSP, computation is grouped onto processors in the program itself. Therefore, any heterogeneous simulation of a BSP program is constrained in its placement and scheduling choices. By choosing to express our programs as dependences, we overcome this and allow full adaptation to the target.

There has also been considerable work on runtimes and languages for parallel systems. StarPU [3,2] uses the HEFT scheduling algorithm [14] to choose an implementation of a task, running it on the relevant processor. However, this framework does not allow choices between sets of tasks.

Coordination languages are perhaps the closest in spirit to our approach. They link together functions from another language to form a complete program. This is similar to how transitions (which are also written in a separate language) are linked together by data places. These include coarse-grain dataflow approaches and are related to streaming languages such as StreamIt [11]. Our model can be seen as a development of both of these.

9 Conclusions

This paper has given a fresh perspective to placement and scheduling on heterogeneous architectures, which we argued in Section 1 are important problems in trying to achieve portable performance. Petri-nets (Section 3) have long been known to provide elegant encodings of concurrency constructs, and we have shown in Section 5 that as an intermediate representation they can also encode the executions of these constructs on our hardware model (Section 4). Whilst the resultant problem is NP-hard, it allows consideration of all performance non-determinism, and there are likely to be efficient heuristics.

In Section 7, we moved away from theory to discuss how compiler optimisations could be performed on our representation.

We feel that the model is promising not just as a theoretical model, but also as the basis of a virtual machine offering portability across heterogeneous architectures. Future work will need to find suitable heuristics for the optimisation problem by further investigation of the related work discussed in Section 6, and

also move towards an implementation that allows investigation of behaviour and performance on real programs.

Acknowledgements. We thank the Schiff Foundation, University of Cambridge, for funding this research through a PhD studentship. Thanks are also due to Jonathan Hayman for valuable discussions and feedback.

References

1. Adve, V., Lattner, C., Brukman, M., Shukla, A., Gaeke, B.: LLVA: A Low-level Virtual Instruction Set Architecture. In: Proc. 36th Annual ACM/IEEE International Symposium on Microarchitecture, MICRO-36 (2003)
2. Augonnet, C., Clet-Ortega, J., Thibault, S., Namyst, R.: Data-Aware Task Scheduling on Multi-Accelerator based Platforms. In: Proc. 16th IEEE International Conference on Parallel and Distributed Systems (ICPADS) (2010)
3. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.-A.: StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In: Sips, H., Epema, D., Lin, H.-X. (eds.) Euro-Par 2009. LNCS, vol. 5704, pp. 863–874. Springer, Heidelberg (2009)
4. Cambini, R., Gallo, G., Scutellà, M.: Flows on Hypergraphs. Mathematical Programming 78, 195–217 (1997)
5. Erlebach, T., Kääb, V., Möhring, R.H.: Scheduling AND/OR-Networks on Identical Parallel Machines. In: Solis-Oba, R., Jansen, K. (eds.) WAOA 2003. LNCS, vol. 2909, pp. 345–346. Springer, Heidelberg (2004)
6. Gillies, D., Liu, J.-S.: Scheduling Tasks with AND/OR Precedence Constraints. In: Proc. 2nd IEEE Symposium on Parallel and Distributed Processing, pp. 394–401 (1990)
7. Goldberg, A.V., Tarjan, R.E.: Finding Minimum-cost Circulations by Successive Approximation. Mathematics of Operations Research 15(3), 430–466 (1990)
8. Jensen, K.: Coloured Petri Nets: A high level language for system design and analysis. In: Rozenberg, G. (ed.) APN 1990. LNCS, vol. 483, pp. 342–416. Springer, Heidelberg (1991)
9. Karp, R.M.: Reducibility Among Combinatorial Problems. In: Complexity of Computer Computations, pp. 85–103. Plenum Press, New York (1972)
10. Pratt, V.: Modeling concurrency with partial orders. International Journal of Parallel Programming 15(1), 33–71 (1986)
11. MIT Computer Architecture Group.: StreamIt, `http://groups.csail.mit.edu/cag/streamit/`
12. Nielsen, M., Plotkin, G., Winskel, G.: Petri nets, Event Structures and Domains, part I. Theoretical Computer Science 13(1), 85–108 (1981)
13. Suleman, M.A., Qureshi, M.K., Khubaib, P.Y.N.: Feedback-directed Pipeline Parallelism. In: Proc. 19th International Conference on Parallel Architectures and Compilation Techniques (PACT), pp. 147–156. ACM, New York (2010)
14. Topcuoglu, H., Hariri, S., Wu, M.-Y.: Performance-effective and Low-complexity Task Scheduling for Heterogeneous Computing. IEEE Transactions on Parallel and Distributed Systems 13(3), 260–274 (2002)
15. Valiant, L.G.: A Bridging Model for Parallel Computation. Communications of the ACM 33, 103–111 (1990)