Robust Software Partitioning
with Multiple Instantiation

Simon A. Spacey, Wolfram Wiesemann, Daniel Kuhn, and Wayne Luk

Department of Computing, Imperial College London, London SW7 2AZ, United Kingdom

simon.spacey@sase.biz, { wwiesema | dkuhn | wl } @ imperial.ac.uk

The purpose of software partitioning is to assign code segments of a given computer program to a range of execution locations such as general purpose processors or specialist hardware components. These execution locations differ in speed, communication characteristics, and in size. In particular, hardware components offering high speed tend to accommodate only few code segments. The goal of software partitioning is to find an assignment of code segments to execution locations that minimizes the overall program run time and respects the size constraints. In this paper we demonstrate that an additional speedup is obtained if we allow code segments to be instantiated on more than one location. We further show that the program run time not only depends on the execution frequency of the code segments but also on their execution order if there are multiply instantiated code segments. Unlike frequency information, however, sequence information is not available at the time when the software partition is selected. This motivates us to formulate the software partitioning problem as a robust optimization problem with decision-dependent uncertainty. We transform this problem to a mixed-integer linear program of moderate size and report on promising numerical results.

Key words: robust optimization; software partitioning; decision-dependent uncertainty; multiple instance partitioning

---

1.   Introduction

We consider a computer program that must be executed quickly and frequently over a long (maybe indefinite) life time. Such programs arise in cryptography (Cheung et al., 2005), digital signal processing (Constantinides et al., 2003), computer vision (Fahmy et al., 2007), video image processing (Haynes et al., 2000), database processing (Shirazi et al., 2001), network analysis (Yusuf et al., 2008) and on-line commercial services. It is assumed that the program consists of several indivisible building blocks or *code segments*. The overall execution time of the program is the time needed to execute the individual code segments and the time needed to exchange information between code segments which are executed in direct succession. These contributions to the run time will be referred to as the *execution costs* and *communication costs*, respectively, and may depend on the characteristics of the *execution location* where each code segment is run. Examples of execution locations are

central processing units, graphical processing units, or specialist hardware components such as field-programmable gate arrays. The goal of software partitioning is to find an assignment of code segments to execution locations that results in the fastest total program execution while respecting the size constraints of specialist hardware components. More precisely, we seek an assignment which ensures that the underlying program runs fast *on average* for a broad range of possible input parameters.

The necessity for software partitioning has been recognized since the early sixties, when manual partitioning methods based on the Fixed Plus Variable system were proposed (Estrin, 2002). By the early nineties, the Cosyma and Vulcan systems (Ernst et al., 1993; Gupta and Micheli, 1993) began to apply semi-automated approaches to software partitioning. While Cosyma relies on a simulated annealing heuristic to assign code segments to execution locations, Vulcan employs a greedy approach. In the following years, a plethora of heuristic solution procedures for software partitioning were studied including tabu search (Eles et al., 1997), genetic algorithms (Dick and Jha, 1997; Purnaprajna et al., 2007), particle swarm algorithms (Abdelhalim et al., 2006) and ant colony optimization (Koudil et al., 2005). Recent exact solution approaches for various forms of the software partitioning problem rely on dynamic programming (Knudsen and Madsen, 1996; Kuang et al., 2005; Shrivastava et al., 2000; Wu and Srikanthan, 2006) and mixed-integer linear programming (Arató et al., 2003; Banerjee et al., 2006; Khayam et al., 2001; Niemann and Marwedel, 1996; Spacey et al., 2009a). A survey on software partitioning and related areas is provided by Wolf (2003).

For the purpose of software partitioning, a program's execution is described exhaustively by the *execution sequence* or *execution trace*, that is, the sequence in which the program's code segments are executed. In typical programs, some code segments are called very often as part of nested loops. This renders the execution trace information too large to be stored at the fine-grained (program basic block or subroutine) level required for optimal software partitioning (Spacey et al., 2009a). Moreover, execution traces often depend on input data implying that there may be infinitely many possible traces that have to be considered for some programs.

As it is impractical or impossible to manipulate large execution traces, software engineers tend to deal with control flow graphs (CFGs) instead. CFGs compress out sequence information from traces and retain only a program's *calling frequencies*, that is, the frequencies with which the code segments call each other. It is relatively easy to predict average calling frequencies over a large number of program runs with statistically independent input data.

Hence, it is reasonable to assume that this type of frequency information is available at the time when the software partition has to be selected.

Frequency information is sufficient to solve the software partitioning problem optimally if each code segment is assigned to a single location (Arató et al., 2003; Khayam et al., 2001; Spacey et al., 2009a). However, in this work we seek to obtain an additional speedup by assigning code segments to more than one execution location. In this multiple instantiation setting, information about the program's execution sequence is required in order to solve the partitioning problem optimally. The motivation for considering multiple instantiation is that in software partitioning, as with most distributed process optimization problems, communications on the same location incur substantially smaller costs than those between different locations. Therefore, it can be beneficial to instantiate frequently visited code segments at more than one location, just as a manufacturer would naturally consider installing the same machine at multiple locations to reduce transportation costs. Previous exact approaches to multiple instantiation have assumed complete knowledge of the program's execution sequence (Banerjee et al., 2006; Niemann and Marwedel, 1996; Kuang et al., 2005). By the above discussion, however, this assumption restricts the applicability of these approaches to programs with short execution traces.

In this paper we propose a novel approach to software partitioning which does not require sequence information but still allows for multiple instantiation of code segments. Since sequence information is absent, we formulate the multiple instance partitioning problem as a robust optimization problem which minimizes the worst-case run time over all execution sequences consistent with known CFG calling frequencies. After applying a dimensionality reduction mechanism, we end up with a robust optimization model with integer decisions and a decision-dependent uncertainty set. We reformulate the resulting model as an equivalent mixed integer linear program (MILP) which we solve with off-the-shelf optimization software.

Although our formulation only requires information about the calling frequencies, its objective function (the worst-case run time) is determined by the *location-aware execution traces* that are consistent with the given calling frequencies. A location-aware execution trace collects full information about the order of the code segment/location pairs visited during program execution, and it is crucially influenced by the adopted *calling convention*. Indeed, whenever a specific code segment must be executed that has been instantiated on several locations, the calling convention decides which of its duplicates is called. While it is straightforward to construct an optimization model that determines the optimal calling
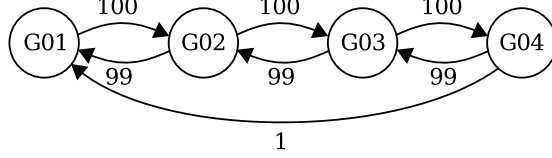
3

Figure 1: Example control flow graph (CFG). The vertices represent the code segments, the arcs depict calls between code segments, and the arc weights indicate calling frequencies.
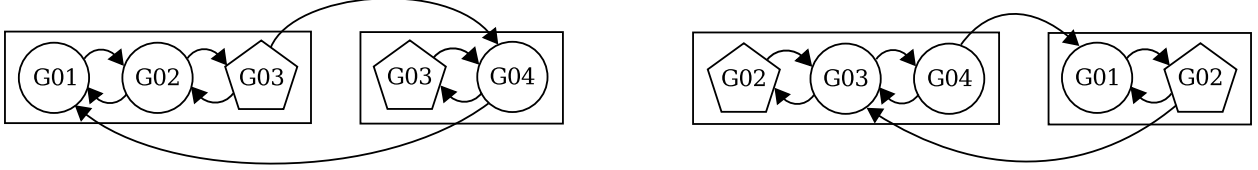

Figure 2: Two software partitions for the CFG from Figure 1. The arcs represent the adopted calling conventions for execution traces τ^1 (left partition) and τ^2 (right partition). Multiply instantiated nodes are shown as pentagons and singly instantiated nodes as circles.

convention when the whole program trace is known, it becomes a major challenge to derive an optimal calling convention if only calling frequencies are available. To ensure computational tractability, we adopt a problem independent *greedy calling convention* in this work.

Example 1.1. *Consider the CFG shown in Figure 1. Among others, this CFG is consistent with the execution trace τ^1 given by*

$$\underbrace{1,2,3,2,\ \ 1,2,3,2,\ \ \ldots,\ \ 1,2,3,2,}_{99\ times}\ \ 1,2,\ \ \underbrace{3,4,\ \ 3,4,\ \ \ldots,\ \ 3,4,}_{100\ times}\ \ 1$$

and the execution trace τ^2 given by

$$\underbrace{1,2,\ \ 1,2,\ \ \ldots,\ \ 1,2,}_{100\ times}\ \ \underbrace{3,4,3,2,\ \ 3,4,3,2,\ \ \ldots,\ \ 3,4,3,2}_{99\ times}\ \ 3,4,1,$$

where the numbers 1–4 represent calls to the code segments G01–G04.

Figure 2 shows two different partitions for the program. The left (right) partition results in 2 (200) expensive cross-partition calls if execution trace τ^1 is realized and in 200 (2) cross-partition calls if execution trace τ^2 is realized. If we remove the first instance of G03 in the left partition, then all code segments are singly instantiated and the partition results in 200 cross-partition calls for either execution trace.

Our robust software partitioning problem bears some similarity to the generalized quadratic assignment problem (GQAP), see Hahn et al. (2008). The GQAP considers a number of entities with given sizes that need to be assigned to locations with size constraints. Every

4

pair of entities gives rise to communication and assignment costs that depend on the locations the entities are assigned to, and the goal is to minimize the overall costs. The software partitioning problem introduced in this paper can be regarded as a multiple instance generalization of GQAP, which is in itself a generalization of the NP-hard quadratic assignment problem (Garey and Johnson, 1979).

We evaluate our approach on a real software benchmark for execution on an architecture with three execution locations. Although the program is fairly small with an execution trace of 2,946 entries for 23 assignable code segments, it is already too large for traditional exact software partitioning approaches. Our numerical experiments demonstrate that the robust multiple instantiation approach may have significant advantages over both an optimistic multiple instantiation model and a single instance GQAP approach in simulated timings for the benchmark's real execution trace.

The remainder of this paper proceeds as follows. In Section 2 we study the software partitioning problem under the assumption that the program's execution trace is precisely known. The problem then reduces to an integer quadratic program that optimizes over all admissible partitions and location-aware execution traces. In Section 3 we argue that, in reality, only frequency information is available at the time when the partition is selected, and that actual sequence information is revealed gradually during program execution. In this more realistic setting, the software partitioning problem reduces to a robust multistage optimization problem with integer recourse and is therefore severely computationally intractable. Section 4 suggests the use of a greedy calling convention and a problem reformulation in terms of location-aware control flows to improve computational tractability. These simplifications lead to a robust single-stage optimization problem with decision-dependent uncertainty. In Section 5 we demonstrate that this robust problem can be reformulated as an equivalent MILP. We provide numerical results in Section 6 and conclude in Section 7.

Notation An arc-weighted directed graph is called (strongly) *connected* if there is a directed path between any two vertices in the graph, and each arc on this connecting path has strictly positive weight. We call a node in an arc-weighted graph *isolated* if it is only incident to arcs of weight zero. We define $\mathbb{B} := \{0, 1\}$.

2. Software Partitioning under Complete Information

We consider a computer program given by a set of code segments $\mathcal{V} := \{1, \ldots, V\}$ and an execution trace $\tau := \{\tau_v\}_{v \in \mathcal{V}}$, where each τ_v represents a function from $\mathbb{T} := \{1, \ldots, T\}$ to $\mathbb{B}$. By definition, $\tau_v(t) := 1$ if code segment v is executed at time t, and $\tau_v(t) := 0$ otherwise. Note that the execution times $t \in \mathbb{T}$ represent time indices rather than factual time points. The factual execution times of the code segments depend on the execution and communication costs and will be discussed later. In the following, we assume that the code segments are executed sequentially, that is, we assume that τ satisfies

$$\sum_{v \in \mathcal{V}} \tau_v(t) = 1 \quad \forall\, t \in \mathbb{T}. \tag{2.1}$$

For situations where code segments may be executed in parallel, one can apply the method presented by Spacey et al. (2009a) to obtain a sequential description of the program which satisfies (2.1). We say that segment v calls segment w at time t if $\tau_v(t) = \tau_w(t+1) = 1$. Without loss of generality, we assume that the program starts with code segment 1 and returns to this first code segment at time $T + 1$, that is, we set $\tau_v(T+1) := \tau_v(1) := 1$ for $v = 1$; $:= 0$ for $v \neq 1$.

A program's execution trace typically depends on the input data, which itself differs with every execution. Since we are interested in the long term program performance over all future inputs, we will not consider a single execution trace. Instead, from now on we assume that τ represents the concatenation of all future execution traces. In this section, we thus assume that complete information about the composite execution trace is available at the time the software partition is selected. This assumption will be relaxed in later sections.

In the following we assume that there is a finite set of possible execution locations $\mathcal{L} := \{1, \ldots, L\}$. The size of location l is given by S_l. We assume that code segment v has size s_{vl} on location l. Note that we use an abstract notion of 'size' which accounts for heterogeneous resource types such as program and data memory requirements as well as physical area for logical gates. Nevertheless, all of the following models extend to multidimensional resource measures in a straightforward way. A software partition can formally be represented as a matrix of binary variables $x \in \mathbb{B}^{V \times L}$ with the interpretation that $x_{vl} = 1$ if and only if segment v is assigned to location l. Partition x is feasible if it is an element of the set

$$X := \left\{ x \in \mathbb{B}^{V \times L} : \sum_{l \in \mathcal{L}} x_{vl} \geq 1 \,\forall v \in \mathcal{V}, \ \sum_{v \in \mathcal{V}} s_{vl} x_{vl} \leq S_l \,\forall l \in \mathcal{L}, \ x_{1l} = \mathbb{I}_{l=1} \right\},$$

where $\mathbb{I}_{l=1} := 1$ if $l = 1$; $:= 0$ otherwise. The first group of constraints in the definition of X requires that each code segment is assigned at least to one location. This guarantees that the program can be executed without errors. The second constraint group enforces the size restrictions on the different locations. Note that we explicitly allow for multiple instantiation of individual code segments. The last constraint requires that the first code segment (with index 1) is instantiated only on location 1. This can always be enforced by introducing a virtual dummy code segment and/or location. For later reference, we also introduce the set $X_1 \subset X$ which only allows for single instantiation. Thus, X_1 is obtained by replacing the inequality in the first constraint group of X by an equality.

If an instance of segment v on location l calls an instance of segment w on location m anytime during program execution, then a calling cost c_{vwlm} is incurred. This cost represents a latency and accounts for delays due to execution of segment v as well as communication between segments v and w. If the instances of v and w reside on different locations, then the communication costs are typically high. Relatively low communication costs arise if the instances of v and w occupy the same location. After data transfer, the execution costs depend solely on the location l where the code segment is executed. In practice it is often impossible to assign all code segments to the location where they incur their lowest execution costs because of hardware size constraints.

Given an assignment $x \in X_1$ subject to single instantiation of the code segments, the overall execution time of the program amounts to

$$c_1(x) := \sum_{t\in\mathbb{T}} \sum_{v,w\in\mathcal{V}} \sum_{l,m\in\mathcal{L}} c_{vwlm} x_{vl}\tau_v(t) x_{wm}\tau_w(t+1).$$

Recall that the program was assumed to return to the first code segment after termination, and observe that the costs associated with this call can be set to zero if necessary. If only single instantiation is allowed, the best software partition is thus found by solving the integer quadratic program

$$\min_{x\in X_1} c_1(x). \qquad (\mathcal{P}_1)$$

Remark 2.1. *Note that problem $\mathcal{P}_1$ encapsulates the quadratic assignment problem as a special case. To see this, set $\mathcal{V} = \mathcal{L} = \{1, \dots, V\}$, $S_l = 1$, $s_{vl} = 1$, $T = V^2 + 1$, and let the trace τ describe an Eulerian cycle in the complete directed graph $(\mathcal{V}, \mathcal{V} \times \mathcal{V})$ with self-loops for all nodes. Recall that an Eulerian cycle in a graph is a cycle that traverses each arc exactly once. Every complete directed graph possesses an Eulerian cycle (Diestel, 2005). We*

thus have $\sum_{t \in \mathbb{T}} \tau_v(t)\tau_w(t+1) = 1$ for all $v, w \in \mathcal{V}$, and problem $\mathcal{P}_1$ reduces to

$$
\begin{aligned}
\min_{x \in \mathbb{B}^{V^2}} \quad & \sum_{v,w,l,m=1}^{V} c_{vwlm} x_{vl} x_{wm} \\
s.t. \quad & \sum_{l=1}^{V} x_{vl} = 1 \quad \forall\, v = 1, \ldots, V, \\
& \sum_{v=1}^{V} x_{vl} \leq 1 \quad \forall\, l = 1, \ldots, V, \\
& x_{11} = 1.
\end{aligned}
\tag{2.2}
$$

Note that since x is a binary square matrix, all inequalities in this problem can be replaced by equalities. Thus, (2.2) is readily recognizable as a variant of the quadratic assignment problem. It is well known that the quadratic assignment problem is strongly NP-hard. The software partitioning problem $\mathcal{P}_1$ and its generalizations to be developed below are thus also strongly NP-hard, that is, they allow for no polynomial-time solution or approximation scheme (Garey and Johnson, 1979).

The situation is further complicated if multiple instantiation of code segments is admissible. To see this, assume that at time t the program executes an instance of segment v on location l (note that there may be other instances of v on locations $l' \neq l$). Moreover, assume that v calls a segment w which is multiply instantiated. Thus, at time $t+1$ the program can jump to one of several locations on which w is instantiated. Note that the given description of the program in terms of the execution trace τ provides no guidelines on which instance to choose. Instead, we are free to adopt any *calling convention* for choosing among the different instances of w.

The additional flexibility to choose a calling convention can be exploited to further reduce the overall execution time of the program. To this end, we introduce a *location-aware execution trace* θ which represents a function from $\mathbb{T}$ to the family of binary matrices $\mathbb{B}^{V \times L}$. By definition, we set $\theta_{vl}(t) = 1$ if and only if code segment v is executed on location l at time t. The location-aware execution trace θ must therefore be an element of the set

$$
\Theta_{\mathrm{PI}}(x; \tau) := \Big\{ \theta \in \mathbb{B}^{V \times L \times T} : \theta_{vl}(t) \leq \tau_v(t)\, x_{vl} \; \forall v \in \mathcal{V}, l \in \mathcal{L}, t \in \mathbb{T},
$$

$$
\sum_{l \in \mathcal{L}} \theta_{vl}(t) = \tau_v(t)\; \forall v \in \mathcal{V},\, t \in \mathbb{T} \Big\}.
$$

The subscript 'PI' indicates that perfect information about the ordinary execution trace τ is assumed to be available. This assumption will be reconsidered in Section 3. The first

constraint group in the definition of $\Theta_{\mathrm{PI}}(x;\tau)$ ensures that an instance of segment v on location l is executed at time t only if v is in fact instantiated on l and *some* instance of v must in fact be executed at time t. The second constraint group makes sure that exactly one instance of segment v at time t is called if v is supposed to be executed at t. Note that our definition of execution traces implies that $\theta_{vl}(T+1) := \theta_{vl}(1)$ for all $v \in \mathcal{V}$ and $l \in \mathcal{L}$, indicating that after termination the program must return to the initial code segment and location. It is easy to verify that for $x \in X_1$ we have $\Theta_{\mathrm{PI}}(x;\tau) = \{\theta^\circ\}$ where $\theta_{vl}^\circ(t) := \tau_v(t)x_{vl}$ for all v, l, and t.

Given an assignment $x \in X$ and a calling convention $\theta \in \Theta_{\mathrm{PI}}(x;\tau)$, the overall execution time of the program amounts to

$$c(\theta) := \sum_{t \in \mathbb{T}} \sum_{v,w \in \mathcal{V}} \sum_{l,m \in \mathcal{L}} c_{vwlm} \theta_{vl}(t) \theta_{wm}(t+1) \,. \tag{2.3}$$

If multiple instantiation of code segments is allowed, the best software partition is thus found by solving the integer quadratic program

$$\min_{x \in X} \ \min_{\theta \in \Theta_{\mathrm{PI}}(x;\tau)} \ c(\theta) \,. \tag{$\mathcal{P}$}$$

Since X_1 is a subset of X, the optimal value of $\mathcal{P}$ is never larger than the optimal value of $\mathcal{P}_1$. In other words, the extra flexibility introduced by allowing for multiple instantiation necessarily reduces the program's optimal execution time.

3. Causal Calling Conventions

A crucial assumption underlying the software partitioning problem $\mathcal{P}$ is that the execution trace τ is precisely known. Note that T represents the total number of executions of all code segments in $\mathcal{V}$ during the program's lifetime. Even for a single program run, T typically exceeds V since some code segments are called several times. Since τ represents the concatenation of all future program traces, T can be expected to be much larger than V. Moreover, τ depends on future input data which is unknown at the time when $\mathcal{P}$ is solved. Since $\mathcal{P}$ requires full trace information as an input, it can therefore not be solved in practice. Instead of collecting, storing and manipulating τ itself, one needs to compress its essential information in an efficient way. This is most commonly achieved by removing sequence information from the trace using a high-level control flow graph (Spacey et al., 2009b), that is, only calling frequencies are gathered. A program's calling frequencies can be estimated with

a number of software packages, such as Valgrind, GILK and 3S (Nethercote and Seward, 2003; Pearce et al., 2002; Spacey, 2006). The basic idea is to amend the program's code with instructions that log every call between different code segments. When the execution trace of a program depends on the input parameters, several program runs can be used to log the cumulative number of calls between code segments. Assuming that the program is executed repeatedly with independent and identically distributed input parameters, the strong law of large numbers then guarantees that the relative calling frequencies converge.

Let us consider the directed, arc-weighted control flow graph $\mathcal{G}$ associated with the program. The vertices of this graph correspond to the code segments $v \in \mathcal{V}$, while the arcs $(v, w) \in \mathcal{V} \times \mathcal{V}$ represent calls between code segments. An arc (v, w) with weight χ_{vw} indicates that segment w is called χ_{vw} times by segment v during program execution. For notational convenience, we assume that $\mathcal{G}$ is complete with self-loops for all nodes. Arcs that do not correspond to calls between code segments are assigned weight zero. Observe that the control flow graph $\mathcal{G}$ is uniquely determined by the execution trace τ. Indeed, the arc weights χ are obtained through the relation

$$\sum_{t \in \mathbb{T}} \tau_v(t)\, \tau_w(t+1) = \chi_{vw} \quad \forall\, v, w \in \mathcal{V}\,. \tag{3.1}$$

In contrast, a given control flow graph $\mathcal{G}$ fails to induce a unique execution trace because it contains no information about the order of the calls. The set of all execution traces consistent with $\mathcal{G}$ is representable as

$$\mathcal{T} := \left\{ \tau \in \mathbb{B}^{V \times T} : \sum_{t \in \mathbb{T}} \tau_v(t)\tau_w(t+1) = \chi_{vw}\ \forall v, w \in \mathcal{V}, \right.$$
$$\left. \sum_{v \in \mathcal{V}} \tau_v(t) = 1\ \forall t \in \mathbb{T},\ \tau_1(1) = 1 \right\}.$$

The first constraint group in the definition of $\mathcal{T}$ enforces consistency with the calling frequencies stipulated in the control flow graph, while the second constraint group ensures that exactly one code segment is executed at any time under trace τ. The last constraint, finally, requires the program to start and terminate at code segment 1. From now on we assume that only $\mathcal{T}$ (or, equivalently, $\mathcal{G}$) is known at the time when the software partition is selected. Thus, there is uncertainty about which trace $\tau \in \mathcal{T}$ will materialize.

Before we formulate the software partitioning problem under trace uncertainty, we should investigate under what conditions on $\mathcal{G}$ the set $\mathcal{T}$ is nonempty (which is a prerequisite for

10

a meaningful optimization model). The following lemma describes the set of control graphs that guarantee non-emptiness of $\mathcal{T}$.

Lemma 3.1. *The set $\mathcal{T}$ of execution traces which are compatible with the control flow graph $\mathcal{G}$ is nonempty if and only if*

(i) $\chi_{vw} \in \mathbb{Z}_+$ *for all $v, w \in \mathcal{V}$;*

(ii) $\mathcal{G}$ *is the union of isolated vertices and a connected subgraph containing vertex 1;*

(iii) $T = \sum_{v,w \in \mathcal{V}} \chi_{vw}$;

(iv) $\sum_{w \in \mathcal{V}} \chi_{vw} - \chi_{wv} = 0$ *for all $v \in \mathcal{V}$.*

Proof. Assume that $\mathcal{T}$ is nonempty, and select some $\tau \in \mathcal{T}$. Then, (i) holds since τ is integer-valued, while (ii)–(iv) hold since τ induces a cycle in $\mathcal{G}$ that starts at vertex 1 and visits arc (v, w) exactly χ_{vw} times for all $v, w \in \mathcal{V}$. Assume now that the assertions (i)–(iv) are satisfied, and consider the directed multigraph $\mathcal{G}(\chi)$ with vertices $\mathcal{V}$ that has χ_{vw} parallel arcs from v to w for all $v, w \in \mathcal{V}$. Condition (iv) guarantees that each vertex in $\mathcal{G}(\chi)$ has equally many incoming as outgoing arcs. Moreover, $\mathcal{G}(\chi)$ is the union of some isolated vertices and a connected subgraph containing vertex 1; this property is inherited from $\mathcal{G}$. Thus, the multigraph $\mathcal{G}(\chi)$ possesses an Eulerian cycle $\{v_t\}_{t \in \mathbb{T}}$ of length T that starts at vertex 1 and visits each arc exactly once. Any such Eulerian cycle can be used to construct a trace $\tau \in \mathcal{T}$ by setting $\tau_v(t) := 1$ if $v = v_t$; $:= 0$ otherwise. $\square$

In the remainder of this paper we will always assume that the conditions (i)–(iv) of Lemma 3.1 are satisfied, implying that $\mathcal{T}$ is in fact nonempty.

Generic calling conventions generating location-aware traces $\theta \in \Theta_{\mathrm{PI}}(x; \tau)$ are not implementable under incomplete information about τ. Indeed, according to the above discussion, only the control flow graph is known at the time when the software partition is selected. Even though the trace τ is initially unknown, it is revealed during program execution, and thus the amount of available information gradually increases: at any time t, the history of the trace up to time t, that is, the sequence of binary vectors $\{\tau(s)\}_{s=1}^{t}$, is available. Causal (or non-anticipative) calling conventions that exploit this online information remain implementable and exhibit considerable flexibility. For more information on the role of non-anticipativity in decision making under uncertainty see e.g. Kall and Wallace (1994).

Set $\Theta_{\mathrm{PI}}(x) := \times_{\tau \in \mathcal{T}} \Theta_{\mathrm{PI}}(x; \tau)$. Thus, every $\theta \in \Theta_{\mathrm{PI}}(x)$ constitutes a collection of location-aware traces $\theta = (\theta_\tau)_{\tau \in \mathcal{T}}$ where $\theta_\tau \in \Theta_{\mathrm{PI}}(x; \tau)$ for each $\tau \in \mathcal{T}$. Any $\theta \in \Theta_{\mathrm{PI}}(x)$ should be interpreted as a decision rule of the following type: if trace τ materializes, then apply the calling convention that generates θ_τ. Note that this decision rule is (usually) only implementable if perfect trace information is available *before* the first call. We can now introduce the set of all location-aware traces that are generated by *causal calling conventions*.

$$\Theta_{\mathrm{C}}(x) := \Big\{ \theta \in \Theta_{\mathrm{PI}}(x) : \theta_\tau(t) = \theta_{\tau'}(t) \ \forall t \in \mathbb{T}, \ \tau, \tau' \in \mathcal{T}$$
$$\text{with } \tau(s) = \tau'(s) \ \forall \, s = 1, \dots, t \Big\}$$

By definition, $\Theta_{\mathrm{C}}(x)$ is a subset of $\Theta_{\mathrm{PI}}(x)$. Thus, any given $\theta \in \Theta_{\mathrm{C}}(x)$ still constitutes a decision rule of the kind described above. This θ is implementable despite the fact that the full trace τ is only known after program termination. Because of the non-anticipativity constraints in the definition of $\Theta_{\mathrm{C}}(x)$, knowledge of the trace history $\tau(1), \dots, \tau(t)$ up to time t is sufficient to implement the time t calling convention yielding $\theta_\tau(t)$. In fact, all $\tau \in \mathcal{T}$ which are indistinguishable up to time t result in the same call at time t.

The above discussion suggests that we should employ causal calling conventions corresponding to location-aware traces $\theta \in \Theta_{\mathrm{C}}(x)$ if τ is uncertain. As no probabilities can be assigned to the different traces in $\mathcal{T}$, it is reasonable to select a software partition $x \in X$ and location-aware trace $\theta \in \Theta_{\mathrm{C}}(x)$ which are optimal in view of the worst-case realization of τ. Ideally, we thus would like to solve the following robust counterpart of problem $\mathcal{P}$.

$$\min_{x \in X} \ \min_{\theta \in \Theta_{\mathrm{C}}(x)} \ \max_{\tau \in \mathcal{T}} \ c(\theta_\tau) \tag{$\mathcal{RP}$}$$

4.   Complexity Reduction

The software partitioning problem $\mathcal{RP}$ represents a multi-stage robust optimization problem with integer recourse and is therefore severely computationally intractable. Moreover, accumulating trace information during program execution is impractical due to excessive storage requirements; see also the discussion at the beginning of Section 3. To reduce the computational complexity of $\mathcal{RP}$, we now apply several approximations.

4.1   Greedy Calling Convention

First, we shrink the set of admissible location-aware traces to a singleton, that is, we stipulate that a specific *greedy calling convention* generating the location-aware trace θ^* must be used.

Instead of using the trace history $\tau(1), \ldots, \tau(t)$ to decide which instance of a code segment should be called at time $t + 1$, this greedy calling convention passes control to the instance that results in the smallest instantaneous calling costs. The software partitioning problem $\mathcal{RP}$ then reduces to

$$\min_{x \in X} \max_{\tau \in \mathcal{T}} c(\theta_\tau^*). \tag{$\mathcal{GRP}$}$$

If the location-aware trace θ^* is an element of $\Theta_{\mathrm{C}}(x)$, then $\mathcal{GRP}$ is more restrictive than the original problem $\mathcal{RP}$ and thus represents a conservative approximation. In order to specify θ^*, we explicitly define the greedy calling convention μ.

$$\mu : X \times \mathcal{V} \times \mathcal{V} \times \mathcal{L} \to \mathcal{L}, \quad \mu(x; v, w, l) := \min \left\{ \arg\min_{m \in \mathcal{L}, x_{wm} = 1} \{c_{vwlm}\} \right\}$$

Note that the arg min-mapping constitutes a set-valued function. For μ to be well-defined, we must prescribe a rule for selecting a unique minimizer if the arg min mapping returns several values. Without loss of generality, we always select the minimizer with the lowest index. For a given software partition x, the greedy calling convention μ has the following property. If code segment v on location l needs to call code segment w, then calling w's instance on location $\mu(x; v, w, l)$ incurs the smallest instantaneous costs. The location-aware trace θ^* generated by the greedy calling convention μ can be constructed recursively. For all $t \in \mathbb{T}$ we set

$$\theta_{\tau, wm}^*(t + 1) := \begin{cases} 1 & \text{if } \tau_w(t + 1) = 1 \text{ and } m = \mu(x; v, w, l) \\ & \quad \text{for } v \text{ and } l \text{ with } \theta_{\tau, vl}^*(t) = 1, \\ 0 & \text{otherwise.} \end{cases}$$

As usual, we use the convention that $\theta_{\tau, vl}^*(T + 1) := \theta_{\tau, vl}^*(1)$ for all $v \in \mathcal{V}$ and $l \in \mathcal{L}$. Note that the recursive construction of θ^* is well-defined since—by definition of X—the first code segment (with index 1) is instantiated only on location 1, while each other code segment is instantiated on at least one location. The location-aware trace θ^* depends on the selected software partition x. To avoid proliferation of subscripts, however, we notationally suppress this dependency.

Lemma 4.1. *The location-aware trace θ^* is an element of $\Theta_{\mathrm{C}}(x)$.*

Proof. We first show that $\theta_\tau^* \in \Theta_{\mathrm{PI}}(x, \tau)$ for any $\tau \in \mathcal{T}$. By construction, $\theta_{\tau, vl}^*(t)$ is binary and vanishes if $x_{vl} = 0$ or $\tau_v(t) = 0$. This implies

$$\theta_{\tau, vl}^*(t) \le \tau_v(t) x_{vl} \quad \forall v \in \mathcal{V}, \ l \in \mathcal{L}, \ t \in \mathbb{T}.$$

13

By induction on time one can show that for any fixed $t \in \mathbb{T}$ there is exactly one code segment v_t and location l_t such that $\theta^*_{\tau,vl}(t) = 1$ if $v = v_t$ and $l = l_t$; $:= 0$ otherwise. This essentially follows from the fact that μ is a single-valued mapping on its entire domain. In particular, notice that $\theta^*_{\tau,vl}(1) = 1$ if and only if $v = l = 1$. This holds because each execution trace in $\mathcal{T}$ starts with code segment 1, which is instantiated only on location 1. Thus, we find

$$\sum_{l \in \mathcal{L}} \theta^*_{\tau,vl}(t) = \tau_v(t) \quad \forall\, v \in \mathcal{V},\ t \in \mathbb{T},$$

implying that θ^*_τ is indeed an element of $\Theta_{\mathrm{PI}}(x,\tau)$. It remains to be shown that θ^* is causal. To this end, notice that $\theta^*_\tau(1)$ is independent of τ, while $\theta^*_\tau(t+1)$ depends only on $\tau(t+1)$ and $\theta^*_\tau(t)$ for all $t \in \mathbb{T}$. Causality thus follows by induction on t. $\qquad\square$

4.2   Location-Aware Control Flows

Problem $\mathcal{GRP}$ is still not suitable for numerical solution. To improve its computational tractability, we should eliminate its explicit dependence on time. To this end, we introduce a set $\Xi_{\mathrm{c}}(x)$ of *location-aware control flows*.

$$\Xi_{\mathrm{c}}(x) := \Big\{ \xi \in \mathbb{Z}_+^{V^2 \times L^2} : \exists\, \tau \in \mathcal{T} \text{ with}$$
$$\xi_{vwlm} = \sum_{t \in \mathbb{T}} \theta^*_{\tau,vl}(t)\theta^*_{\tau,wm}(t+1)\ \forall v, w \in \mathcal{V},\ l, m \in \mathcal{L} \Big\}$$

The component ξ_{vwlm} of any $\xi \in \Xi_{\mathrm{c}}(x)$ indicates how often code segment v on location l calls code segment w on location m during program execution when the greedy calling convention is employed. If some of the code segments are multiply instantiated, this number may vary with the execution trace τ. The set $\Xi_{\mathrm{c}}(x)$ collects all location-aware control flows ξ associated with the possible traces $\tau \in \mathcal{T}$. In other words, $\Xi_{\mathrm{c}}(x)$ is the set of all location-aware control flows that are consistent with the (location-*un*aware) control flow graph $\mathcal{G}$. Recalling the definition of the cost function (2.3), problem $\mathcal{GRP}$ can now be reformulated as

$$\min_{x \in X} \max_{\xi \in \Xi_{\mathrm{c}}(x)} \sum_{v,w \in \mathcal{V}} \sum_{l,m \in \mathcal{L}} c_{vwlm}\xi_{vwlm} \,. \tag{4.1}$$

Note that (4.1) can be interpreted as a robust optimization problem with *decision-dependent uncertainty*. The goal is to find a robust software partition x that minimizes the worst-case program execution time. The worst case is taken over all location-aware control flows ξ that are consistent with the control flow graph $\mathcal{G}$ and the software partition x. In robust

optimization terminology, x is the decision variable, ξ is the uncertain parameter, and $\Xi_c(x)$ represents the underlying *uncertainty set*. The uncertainty set explicitly depends on the decision x. While stochastic programs with decision-dependent uncertainty have been considered recently by Goel and Grossmann (2006), it seems that robust optimization problems of this type have received little attention until now. See Ben-Tal et al. (2009) for a textbook introduction to robust optimization.

Problem (4.1) constitutes an exact reformulation of $\mathcal{GRP}$. While its objective function is linear in ξ and thus lends itself to computational treatment, the uncertainty set $\Xi_c(x)$ looks cumbersome and still exhibits an explicit dependence on time. We now construct a more tractable approximation for $\Xi_c(x)$. To this end, we let M be any constant which is larger than $\max_{v,w} \chi_{vw}$, and we define $\Xi(x)$ as the set of all $\xi \in \mathbb{R}_+^{V^2 \times L^2}$ satisfying the constraints

$$\sum_{w \in \mathcal{V}} \sum_{m \in \mathcal{L}} \xi_{vwlm} - \xi_{wvml} = 0 \tag{4.2a}$$

$$\sum_{l,m \in \mathcal{L}} \xi_{vwlm} = \chi_{vw} \tag{4.2b}$$

$$\xi_{vwlm} \leq M \min\{x_{vl}, x_{wm}\} \tag{4.2c}$$

$$\xi_{vwlm} \leq M \min_{m' \in \mathcal{L}_{vwlm}} (1 - x_{wm'}) \tag{4.2d}$$

for all $v, w \in \mathcal{V}$ and $l, m \in \mathcal{L}$. The index set $\mathcal{L}_{vwlm}$ is defined as the collection of all $m' \in \mathcal{L}$ that satisfy $c_{vwlm'} < c_{vwlm}$. Notice that $\Xi(x)$ is indeed independent of the choice of M as long as M is larger than all χ_{vw}. We emphasize that $\Xi_c(x)$ is a discrete set, whereas $\Xi(x)$ constitutes a convex polyhedron.

Proposition 4.2. $\Xi_c(x)$ *is a subset of* $\Xi(x)$.

Proof. Choose an arbitrary $\xi \in \Xi_c(x)$ and let τ be an element of $\mathcal{T}$ satisfying

$$\xi_{vwlm} = \sum_{t \in \mathbb{T}} \theta^*_{\tau,vl}(t)\theta^*_{\tau,wm}(t+1) \quad \forall v, w \in \mathcal{V}, \, l, m \in \mathcal{L} \, .$$

The existence of such a τ is guaranteed by the definition of $\Xi_c(x)$. Thus, we have

$$\sum_{w \in \mathcal{V}} \sum_{m \in \mathcal{L}} \xi_{vwlm} - \xi_{wvml}$$

$$= \sum_{w \in \mathcal{V}} \sum_{m \in \mathcal{L}} \sum_{t \in \mathbb{T}} \theta^*_{\tau,vl}(t)\theta^*_{\tau,wm}(t+1) - \theta^*_{\tau,wm}(t)\theta^*_{\tau,vl}(t+1)$$

$$= \sum_{t \in \mathbb{T}} \theta^*_{\tau,vl}(t) - \theta^*_{\tau,vl}(t+1) = \theta^*_{\tau,vl}(1) - \theta^*_{\tau,vl}(T+1) = 0 \, ,$$

15

where the second equality holds since the program executes exactly one code segment on exactly one location at each time. Thus, (4.2a) holds. Next, we find

$$\sum_{l,m\in\mathcal{L}} \xi_{vwlm} = \sum_{l,m\in\mathcal{L}} \sum_{t\in\mathbb{T}} \theta^*_{\tau,vl}(t)\theta^*_{\tau,wm}(t+1) = \sum_{t\in\mathbb{T}} \tau_v(t)\tau_w(t+1) = \chi_{vw},$$

where the second equality holds because $\theta^*_\tau \in \Theta_{\mathrm{PI}}(x;\tau)$, while the third equality follows from the properties of traces $\tau \in \mathcal{T}$. Thus, (4.2b) is established. The fact that θ^*_τ is contained in $\Theta_{\mathrm{PI}}(x;\tau)$ further implies

$$\xi_{vwlm} = \sum_{t\in\mathbb{T}} \theta^*_{\tau,vl}(t)\theta^*_{\tau,wm}(t+1) \leq \sum_{t\in\mathbb{T}} x_{vl}\tau_v(t)x_{wm}\tau_w(t+1) = x_{vl}x_{wm}\chi_{vw}.$$

Thus, we have $\xi_{vwlm} \leq Mx_{vl}$ and $\xi_{vwlm} \leq Mx_{wm}$, which implies (4.2c). In order to establish (4.2d), we notice that

$$
\begin{aligned}
\exists m' \in \mathcal{L}_{vwlm} : x_{wm'} = 1 \;\; &\Rightarrow\;\; \mu(x;v,w,l) \neq m \\
&\Rightarrow\;\; \theta^*_{\tau,vl}(t)\theta^*_{\tau,wm}(t+1) = 0 \quad \forall t \in \mathbb{T} \\
&\Rightarrow\;\; \xi_{vwlm} = \sum_{t\in\mathbb{T}} \theta^*_{\tau,vl}(t)\theta^*_{\tau,wm}(t+1) = 0.
\end{aligned}
$$

Here, the first and second implications follow from the definitions of $\mu(x;v,w,l)$ and the location-aware trace θ^*, respectively. The above reasoning implies that $\xi_{vwlm} \leq M(1-x_{wm'})$ for all $m' \in \mathcal{L}_{vwlm}$. In summary, all constraints (4.2) are satisfied, and thus ξ is an element of $\Xi(x)$. $\qquad\square$

In the following, we argue that $\Xi_{\mathrm{c}}(x)$ is a strict subset of $\Xi(x)$, and we establish an explicit criterion to decide whether $\xi \in \Xi(x)$ is contained in $\Xi_{\mathrm{c}}(x)$. To this end, we assign to each $V^2 \times L^2$-dimensional vector ξ with nonnegative integer entries a directed multigraph $\mathcal{G}(\xi)$ with vertices $\mathcal{V} \times \mathcal{L}$ and with ξ_{vwlm} parallel arcs from (v,l) to (w,m), where (v,l) and (w,m) range over $\mathcal{V} \times \mathcal{L}$.

Proposition 4.3. *If $\xi \in \Xi(x)$ has only integer entries, while $\mathcal{G}(\xi)$ is the union of a connected subgraph and some isolated vertices, then $\xi \in \Xi_{\mathrm{c}}(x)$.*

Proof. Select a ξ satisfying the conditions in the statement. Since ξ is an element of $\Xi(x)$, the number of incoming arcs equals the number of outgoing arcs in each vertex of the multigraph $\mathcal{G}(\xi)$, see (4.2a). Since $\mathcal{G}(\xi)$ can be decomposed into a connected subgraph and some isolated vertices, there exists an Eulerian cycle $\{v_t, l_t\}_{t\in\mathbb{T}}$ of length

$$\sum_{v,w\in\mathcal{V}} \sum_{l,m\in\mathcal{L}} \xi_{vwlm} \overset{(4.2b)}{=} \sum_{v,w\in\mathcal{V}} \chi_{vw} = T$$
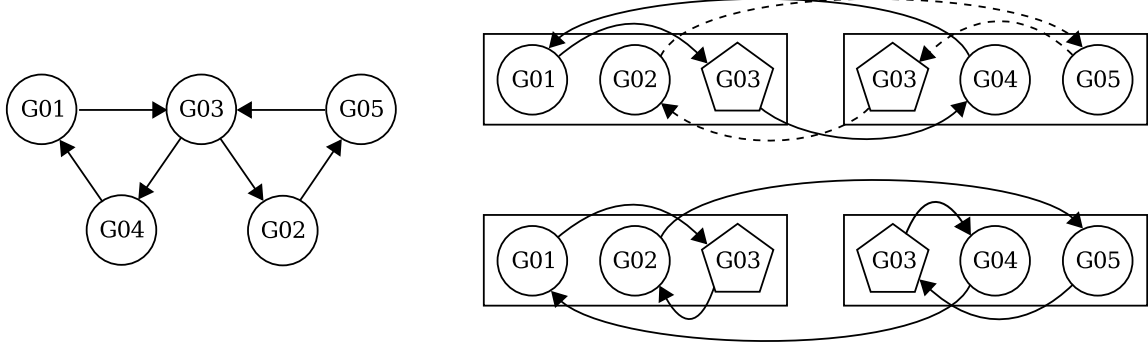
16

Figure 3: Instance of $\mathcal{AGRP}$ with disconnected worst-case control flow. The left chart illustrates the CFG. The upper right and lower right charts depict the worst-case disconnected and worst-case connected control flows for the optimal partition, respectively. In the charts, all arcs have unit weights.

which visits each arc exactly once. The component sequence $\{v_t\}_{t\in\mathbb{T}}$ can be used to construct an execution trace $\tau \in \mathcal{T}$ by setting $\tau_v(t) := 1$ if $v = v_t$; $:= 0$ otherwise.

The constraints (4.2c)–(4.2d) ensure that if $v_t = v$, $v_{t+1} = w$, $l_t = l$, and $l_{t+1} = m$ for some $t \in \mathbb{T}$, then v must be instantiated on l, w must be instantiated on m, and the cost of calling w on m from v on l is minimal over all locations m' on which w is instantiated. Therefore, we have $l_t = \sum_{l\in\mathcal{L}} l\theta^*_{\tau,vl}(t)$ for all $t \in \mathbb{T}$, $v \in \mathcal{V}$, and $l \in \mathcal{L}$, that is, the Eulerian cycle $\{v_t, l_t\}_{t\in\mathbb{T}}$ is induced by the execution trace τ under the location-aware trace θ^*. This implies that $\xi \in \Xi_c(x)$. $\qquad\square$

If we replace $\Xi_c(x)$ by its superset $\Xi(x)$ in (4.1), we obtain a conservative approximation for $\mathcal{GRP}$.

$$\min_{x\in X} \max_{\xi\in\Xi(x)} \sum_{v,w\in\mathcal{V}} \sum_{l,m\in\mathcal{L}} c_{vwlm}\xi_{vwlm} \qquad (\mathcal{AGRP})$$

Note that the inner maximization problem is now a linear program. In the next section we will show that $\mathcal{AGRP}$ has an equivalent reformulation as a MILP and is thus a promising candidate for numerical solution. We close this section with an example which illustrates that the worst-case control flow in $\mathcal{AGRP}$ for a given partition $x \in X$ may indeed be contained in $\Xi(x) \setminus \Xi_c(x)$. We remark, however, that disconnected worst-case control flows seem to be rare in realistic examples.

Example 4.4. *Consider a problem with two execution locations of size 21 and 7, respectively, and five code segments of size 10, 10, 1, 3 and 3, respectively, independent of the execution location. The calling costs are $c_{vwlm} = 1$ if $l = m$ and 10 otherwise, for all*

17

$v, w \in \mathcal{V}$. *Figure 3 shows the CFG (left chart) and the optimal solution to $\mathcal{AGRP}$ (right charts). For this partition, the worst-case control flow is disconnected and leads to cumulative calling costs of 42 (upper right chart), whereas the worst connected control flow leads to cumulative calling costs of 24 (lower right chart).*

5. MILP Formulation

We convert $\mathcal{AGRP}$ to an equivalent minimization problem by dualizing the linear program over the uncertain parameters $\xi \geq 0$. To do so, we introduce Lagrange multipliers α_{vl} and β_{vw} corresponding to the flow conservation and consistency constraints (4.2a) and (4.2b), respectively. Moreover, we assign nonnegative multipliers γ_{vwlm} and δ_{vwlm} to the constraints (4.2c) and (4.2d), respectively, which ensure that the location-aware control flow is consistent with the selected software partition x and obeys the greedy calling convention. The dual of the inner maximization problem adopts the following form.

$$
\begin{aligned}
\min_{\alpha,\beta,\gamma,\delta,\varepsilon} \quad & \sum_{v,w \in \mathcal{V}} \chi_{vw}\beta_{vw} + M \sum_{v,w \in \mathcal{V}} \sum_{l,m \in \mathcal{L}} \min\{x_{vl}, x_{wm}\}\gamma_{vwlm} \\
& + M \sum_{v,w \in \mathcal{V}} \sum_{l,m \in \mathcal{L}} \min_{m' \in \mathcal{L}_{vwlm}} (1 - x_{wm'})\delta_{vwlm} \\
\text{s.t.} \quad & \alpha_{vl} - \alpha_{wm} + \beta_{vw} + \gamma_{vwlm} + \delta_{vwlm} \geq c_{vwlm} \quad \forall v, w \in \mathcal{V},\ l, m \in \mathcal{L} \\
& \gamma,\, \delta \geq 0
\end{aligned}
\tag{5.1}
$$

Notice that strong linear programming duality holds since the inner maximization problem in $\mathcal{AGRP}$ is feasible, that is, because $\Xi(x)$ is nonempty for all $x \in X$. This is a consequence of the fact that $\Xi(x)$ is a superset of $\Xi_{\mathrm{c}}(x)$, which in turn is nonempty because of Lemma 3.1 and our assumptions about the given control flow graph $\mathcal{G}$. Thus, the dual linear program (5.1) has the same optimal value as the inner maximization problem in $\mathcal{AGRP}$.

From a computational point of view, the frequent occurrence of the large constant M in the dual objective function is undesirable as it deteriorates the problem's scaling properties. Moreover, the bilinear terms in the assignment variable x and the dual variables γ and δ lead to a mixed-integer *non*linear program when (5.1) is substituted into $\mathcal{AGRP}$. It turns out that the outlined deficiencies can be overcome by exploiting the following observation. The primal feasible set $\Xi(x)$ of the inner problem in $\mathcal{AGRP}$ is independent of the choice of M as long as M is larger than $M_0 := \max_{v,w} \chi_{vw}$. Thus, by strong duality, the optimal value of (5.1) is also independent of M as long as $M \geq M_0$. This implies that

$$
\sum_{v,w \in \mathcal{V}} \sum_{l,m \in \mathcal{L}} \min\{x_{vl}, x_{wm}\}\gamma_{vwlm} + \min_{m' \in \mathcal{L}_{vwlm}} (1 - x_{wm'})\delta_{vwlm} = 0
\tag{5.2}
$$

at optimality. Since γ and δ are nonnegative, while x is a vector of binary variables, (5.2) can be interpreted as a complementarity condition which is equivalent to

$$\gamma_{vwlm} \leq M_{\mathrm{d}} \max\{1 - x_{vl}, 1 - x_{wm}\}, \ \delta_{vwlm} \leq M_{\mathrm{d}} \max_{m' \in \mathcal{L}_{vwlm}} x_{wm'} \tag{5.3}$$

for all $v, w \in \mathcal{V}$ and $l, m \in \mathcal{L}$. The new constant $M_{\mathrm{d}} > 0$ represents a uniform a priori bound on the optimal dual variables γ and δ with respect to the maximum norm. Note that M_{d} can be chosen independently of $x \in X$ and $M \geq M_0$. This reasoning shows that we can remove all terms proportional to M in the objective of (5.1) at the cost of appending the constraints (5.3). By construction, the optimal value of the resulting streamlined optimization problem is independent of M_{d} as long as this constant is chosen sufficiently large.Using standard duality arguments, one can for example show that the choice $M_{\mathrm{d}} := VL \max\{c_{vwlm} : v, w \in \mathcal{V}, l, m \in \mathcal{L}\}$ is sufficient. Standard arguments similar to those outlined above can be used to show that the constraint

$$\alpha_{vl} - \alpha_{wm} + \beta_{vw} + \gamma_{vwlm} + \delta_{vwlm} \geq c_{vwlm} \tag{5.4}$$

is redundant (that is, not binding at optimality) if $x_{vl} = 0$ or $x_{wm} = 0$ or $x_{wm'} = 1$ for at least one $m' \in \mathcal{L}_{vwlm}$. This observation allows us to eliminate δ from the problem and to replace (5.3) and (5.4) by

$$\alpha_{vl} - \alpha_{wm} + \beta_{vw} + \gamma_{vwlm} \geq c_{vwlm}$$

$$\gamma_{vwlm} \leq M_{\mathrm{d}} \left(1 - x_{vl} + 1 - x_{wm} + \sum_{m' \in \mathcal{L}_{vwlm}} x_{wm'} \right).$$

In summary, we have thus demonstrated that $\mathcal{AGRP}$ can be equivalently expressed as the following MILP.

$$
\begin{aligned}
\min_{x,\alpha,\beta,\gamma} \quad & \sum_{v,w \in \mathcal{V}} \chi_{vw} \beta_{vw} \\
\text{s.t.} \quad & \alpha_{vl} - \alpha_{wm} + \beta_{vw} + \gamma_{vwlm} \geq c_{vwlm} && \forall v, w \in \mathcal{V}, \quad l, m \in \mathcal{L} \\
& \gamma_{vwlm} \leq M_{\mathrm{d}} \left(2 - x_{vl} x_{wm} + \sum_{m' \in \mathcal{L}_{vwlm}} x_{wm'} \right) && {}^{\prime\prime} \\
& x \in X, \ \gamma \geq 0
\end{aligned} \tag{5.5}
$$

The solution time of problem (5.5) is determined by the $\mathcal{O}(VL)$ binary variables. Since $V \gg L$ in typical applications, every additional execution location leads to V further binary variables, and hence the number of execution locations L dominates the solution time.

6. Numerical Results

We compare our robust software partitioning approach with single instantiation partitioning and optimistic multiple instantiation partitioning. More precisely, we evaluate the following approaches to software partitioning:

1. **Robust Multiple Instantiation Software Partitioning.** We solve model $\mathcal{AGRP}$.

2. **Optimistic Multiple Instantiation Software Partitioning.** We replace the inner maximization in $\mathcal{AGRP}$ by a minimization. The resulting problem $\mathcal{OP}$ optimizes in view of the *least* time-consuming control flow consistent with the given frequency information. We also consider a variant $\mathcal{OP}^*$ that enforces connectivity of the location-aware control flow ξ. Connectivity can be enforced through subtour elimination constraints known from the traveling salesman literature (Applegate et al., 2007).

3. **Single Instantiation Software Partitioning.** We consider model $\mathcal{P}_1$ from Section 2, where every code segment must be assigned to exactly one location. Since the program's execution time is uniquely determined by the calling frequencies and does not depend on the execution sequence, $\mathcal{P}_1$ reduces to a generalized quadratic assignment problem (Hahn et al., 2008).

The optimal values of the aforementioned partitioning approaches satisfy

$$\mathcal{OP} \preceq \mathcal{OP}^* \preceq \mathcal{P} \preceq \mathcal{AGRP} \preceq \mathcal{P}_1,$$

where $\mathcal{P}$ denotes the software partitioning problem under complete information (see Section 2) and '$\preceq$' refers to the ordering of optimal objective values. The first inequality follows from the fact that subtour elimination constraints reduce the feasible region of the inner optimization problem in $\mathcal{OP}$. The last inequality holds since $\mathcal{AGRP}$ minimizes over the set of multiple instantiation partitions, while $\mathcal{P}_1$ optimizes over the smaller set of single instantiation partitions.

We apply all software partitioning approaches to a Java simulation program. Figure 4 illustrates the control flow graph for a representative program run to be optimized. The graph is obtained with the 3S characterization framework discussed by Spacey (2006). For the sake of clarity, we omit the auxiliary arc that connects the sink node with the source node throughout this section. We consider three heterogeneous execution locations A, B and C.
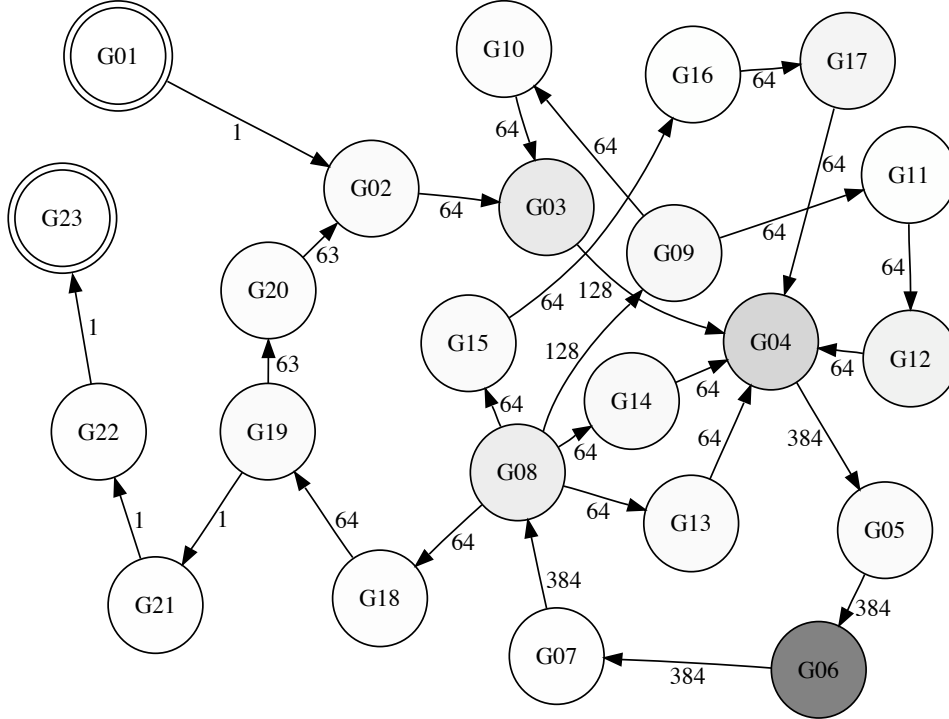
Figure 4: Control flow graph of the software benchmark. Nodes are shaded in proportion to their total inbound communication and computation time requirements on a reference hardware location with darker shades indicating code segments with greater time requirements. G01 and G23 are unique source and sinks for the control flow.

The execution and communication costs of the code segments on the different locations were obtained with the Write-Only Architecture computation model (Spacey et al., 2009a,b). We use the commercial solver CPLEX 11.2 to solve the arising MILP models.

Table 1 compares the solutions of the considered partitioning approaches in terms of their forecasted and simulated execution times, the number of duplicate code segments, as well as the solution times required by CPLEX. The forecasted execution times correspond to the optimal objective values of the corresponding optimization problems. For the models $\mathcal{AGRP}$ and $\mathcal{OP}$ this is the worst-case and best-case execution time over all location-aware control flows that are consistent with the control flow graph in Figure 4, respectively. For the model $\mathcal{P}_1$ this is the execution time for any location-aware control flow that is consistent with the control flow graph. The simulated execution times are obtained with a 3S simulation of the real program execution trace τ containing 2,946 calls to the code segments. In the simulations we use the assignments $x \in X$ determined by the optimization models and implement a greedy calling convention. The number of duplicate code segments represents

the difference between the overall number of instantiations and $|\mathcal{V}|$, the number of instantiations in $\mathcal{P}_1$. As expected, the optimistic model underestimates the factual execution time. Due to its inherent optimism, the model determines a partition with many duplicate code segments that performs well for some benign execution flows but performs poorly on average. The single instantiation model, on the other hand, correctly predicts the factual execution times but determines a poor partition due to the single instantiation restriction. The robust partitioning approach, finally, outperforms both methods in the simulations because it predicts the factual execution times more accurately and allows for multiple instantiation. Note that the number of duplicates in the optimal solution for model $\mathcal{AGRP}$ is significantly smaller than the respective numbers for $\mathcal{OP}$ and $\mathcal{OP}^*$. Indeed, duplicates have two opposite effects on the uncertainty set $\Xi(x)$: constraint (4.2c) is relaxed, whereas constraint (4.2d) is tightened. Thus, contrary to the optimistic models, more duplicates do not necessarily lead to better solutions in $\mathcal{AGRP}$.

We remark that the increased performance of model $\mathcal{AGRP}$ comes at the cost of significantly larger solution times. Since the partitioning problem needs to be solved only once at design time, the increase in solution time might well be acceptable in view of the potential performance gains. Alternatively, the solution times for model $\mathcal{AGRP}$ could be reduced in two ways. On one hand, one could design integrality cuts for $\mathcal{AGRP}$ that speed up the fathoming process of CPLEX. On the other hand, one could resort to heuristic solution techniques, for example problem decomposition methods, a tabu search or a genetic algorithm. Depending on their solution times, these heuristics may even be used to dynamically adapt the software partition while the program is executed (Hauck and Dehon, 2008).

Figures 5–8 illustrate the solutions obtained by the different partitioning approaches. Circles indicate code segments that are instantiated once, while pentagons refer to multiply instantiated code segments. The arcs display the location-aware control flows that determine the corresponding objective values. As Figure 6 shows, $\mathcal{OP}$ selects a partition x that minimizes the overall execution costs for a disconnected (and hence physically impossible) control flow. While this leads to a poor practical performance, the partition is nevertheless feasible since it is an element of the set X. In particular, the partition contains at least one instance of each code segment, and for any possible control flow, the greedy calling convention will determine a valid code segment to call. The refined model $\mathcal{OP}^*$, on the other hand, enforces connectivity of the location-aware control flows ξ.

model	execution times (secs)		# of duplicates	solution time
	forecasted	simulated		
$\mathcal{AGRP}$	479	479	7	26h:48m:53s
$\mathcal{OP}$	167	871	12	00h:00m:01s
$\mathcal{OP}^*$	321	853	10	00h:05m:15s
$\mathcal{P}_1$	1037	1037	0	00h:00m:24s

Table 1: Summary of the solutions for the benchmark instance from Figure 4.

7. Conclusion

Previous exact approaches to software partitioning either assume knowledge of the complete execution sequence or they only support the single instantiation of code segments. In practice, sequence information is available only at coarse granularity levels which reduces assignment optimization potentials (Spacey et al., 2009a), while the restriction to single instantiation partitions can severely reduce the achievable program performance, see e.g. Table 1. In this paper we present a novel approach to software partitioning with multiple instantiation that only requires knowledge of the control flow graph, which is stripped of all sequence information.

As soon as code segments can be instantiated multiple times, the execution time of an assignment depends not only on the calling frequencies but also on the execution sequence. In the absence of such sequence information, we propose to formulate the multiple instantiation software partitioning problem as a robust optimization problem that minimizes the worst-case run time over all execution traces consistent with the known control flow graph. We show that the resulting problem can be approximated by a MILP amenable to optimization with off-the-shelf commercial solvers. We also provide results for a benchmark application demonstrating that our approach compares favourably with alternative software partitioning methods when evaluated on real execution traces.

We identify three promising areas for future research. Firstly, even though our method does not require sequence information, it becomes computationally challenging for large applications. In order to improve the scalability of our method, we propose the investigation of new formulations and bounding techniques along the lines of Hahn et al. (2008). Secondly, the robust solution may be too conservative in some instances and the inclusion of additional software characterization information such as compressed partial trace sequences generated by the 3S `loopgraph_d` tool (Spacey, 2006) may be investigated as future work.

Thirdly, research into the effect of different non-anticipative calling conventions and calling cost variation over the uncertainty set $\Xi_c(x)$ should be performed (Spacey, 2009).

Finally, as our robust partitioning approach can be regarded as an extension of the generalized quadratic assignment problem to multiple instances, and as the latter problem has manifold practical applications (Hahn et al., 2008), it seems promising to investigate the applicability of our model to domains outside the software partitioning arena.

References

Abdelhalim, M. B., A. E. Salama, S. E. D. Habib. 2006. Hardware software partitioning using particle swarm optimization technique. *Proceedings of the 6th International Workshop on System-on-Chip for Real-Time Applications*. 189–194.

Applegate, D. L., R. E. Bixby, V. Chvatal, W. J. Cook. 2007. *The Traveling Salesman Problem: A Computational Study*. Princeton University Press.

Arató, P., S. Juhász, Z. A. Mann, A. Orbán, D. Papp. 2003. Hardware-software partitioning in embedded system design. *Proceedings of the 2003 IEEE International Symposium on Intelligent Signal Processing*. 197–202.

Banerjee, S., E. Bozorgzadeh, N. D. Dutt. 2006. Integrating physical constraints in HW-SW partitioning for architectures with partial dynamic reconfiguration. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* **14** 1189–1202.

Ben-Tal, A., A. Nemirovski, L. El Ghaoui. 2009. *Robust Optimization*. Princeton University Press.

Cheung, R. C. C., N. J. Telle, W. Luk, P. Y. K. Cheung. 2005. Customizable elliptic curve cryptosystems. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* **13** 1048–1059.

Constantinides, C. A., P. Y. K. Cheung, W. Luk. 2003. Wordlength optimization for linear digital signal processing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **22** 1432–1442.

Dick, R. P., N. K. Jha. 1997. Mogac: A multiobjective genetic algorithm for the co-synthesis of hardware-software embedded systems. *Proceedings of the 1997 IEEE/ACM International Conference on Computer-Aided Design*. 522–529.

Diestel, R. 2005. *Graph Theory*. 3rd ed. Springer.

Eles, P., Z. Peng, K. Kuchcinski, A. Doboli. 1997. System level hardware/software partitioning based on simulated annealing and tabu search. *Design Automation for Embedded Systems* **2** 5–32.

Ernst, R., J. Henkel, T. Benner. 1993. Hardware-software cosynthesis for microcontrollers. *IEEE Design & Test of Computers* **10** 64–75.

Estrin, G. 2002. Reconfigurable computer origins: The UCLA fixed-plus-variable (F+V) structure computer. *IEEE Computer* **24** 3–9.

Fahmy, S. A., C.-S. Bouganis, P. Y. K. Cheung, W. Luk. 2007. Real-time hardware acceleration of the trace transform. *Journal of Real-Time Image Processing* **2** 235–248.

Garey, M. R., D. S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman.

Goel, V., I.E. Grossmann. 2006. A class of stochastic programs with decision dependent uncertainty. *Mathematical Programming* **108** 355–394.

Gupta, R., G. De Micheli. 1993. Hardware-software co-synthesis for digital systems. *IEEE Design & Test of Computers* **10** 29–41.

Hahn, P. M., B.-J. Kim, M. Guignard, J. M. Smith, Y.-R. Zhu. 2008. An algorithm for the generalized quadratic assignment problem. *Computational Optimization and Applications* **40** 351–372.

Hauck, S., A. Dehon. 2008. *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. Morgan Kaufmann.

Haynes, S. D., J. Stone, P. Y. K. Cheung, W. Luk. 2000. Video image processing with the sonic architecture. *IEEE Computer* **33** 50–57.

Kall, P., S. W. Wallace. 1994. *Stochastic Programming*. John Wiley & Sons.

Khayam, S. A., S. A. Khan, S. Sadiq. 2001. A generic integer programming approach to hardware/software codesign. *Proceedings of the IEEE International Multi Topic Conference*. 6–9.

Knudsen, P. V., J. Madsen. 1996. PACE: A dynamic programming algorithm for hardware/software partitioning. *Proceedings of the 4th International Workshop on Hardware/Software Co-Design*. 85–92.

Koudil, M., K. Benatchba, S. Gharout, N. Hamani. 2005. Solving partitioning problem in codesign with ant colonies. *Artificial Intelligence and Knowledge Engineering Applications: A Bioinspired Approach*. Springer.

Kuang, S.-R., C.-Y. Chen, R.-Z. Liao. 2005. Partitioning and pipelined scheduling of embedded system using integer linear programming. *Proceedings of the 11th International Conference on Parallel and Distributed Systems*. 37–41.

Nethercote, N., K. Seward. 2003. Valgrind: A program supervision framework. *Proceedings of the 3rd Workshop on Runtime Verification*.

Niemann, R., P. Marwedel. 1996. Hardware/software partitioning using integer programming. *Proceedings of the 1996 European Conference on Design and Test*. 473–479.

Pearce, D.J., P.H.J. Kelly, T. Field, U. Harder. 2002. Gilk: A dynamic instrumentation tool for the linux kernel. *Proceedings of the 12th International Conference on Computer Performance Evaluation, Modelling Techniques and Tools*. 220–226.

Purnaprajna, M., M. Reformat, W. Pedrycz. 2007. Genetic algorithms for hardware-software partitioning and optimal resource allocation. *Journal of Systems Architecture: the EU-ROMICRO Journal* **53** 339–354.

Shirazi, N., D. Benyamin, W. Luk, P.Y.K. Cheung, S. Guo. 2001. Quantitative analysis of FPGA-based database searching. *The Journal of VLSI Signal Processing* **28** 85–96.

Shrivastava, A., H. Kumar, S. Kapoor, S. Kumar, M. Balakrishnan. 2000. Optimal hardware/software partitioning for concurrent specification using dynamic programming. *Proceedings of the 13th International Conference on VLSI Design*. 110–113.

Spacey, S. A. 2006. 3S: Program instrumentation and characterisation framework. Tech. rep., Imperial College London.

Spacey, S. A. 2009. Computational partitioning for heterogeneous systems. Ph.D. thesis, Imperial College London.

Spacey, S. A., W. Luk, P. H. J. Kelly, D. Kuhn. 2009a. Coarse-grained parallel partitioning through fine-grained sequential assignment for heterogeneous systems. Working Paper.

Spacey, S. A., W. Luk, P. H. J. Kelly, D. Kuhn. 2009b. Rapid design space visualisation through hardware/software partitioning. *Proceedings of the 5th IEEE Southern Programmable Logic Conference*. 159–164.

Wolf, W. 2003. A decade of hardware/software codesign. *IEEE Computer* **36** 38–43.

Wu, J., T. Srikanthan. 2006. Low-complex dynamic programming algorithm for hardware/software partitioning. *Information Processing Letters* **98** 41–46.

Yusuf, S., W. Luk, M. Sloman, N. Dulay, E. C. Lupu, G. Brown. 2008. Reconfigurable architecture for network flow analysis. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* **16** 57–65.
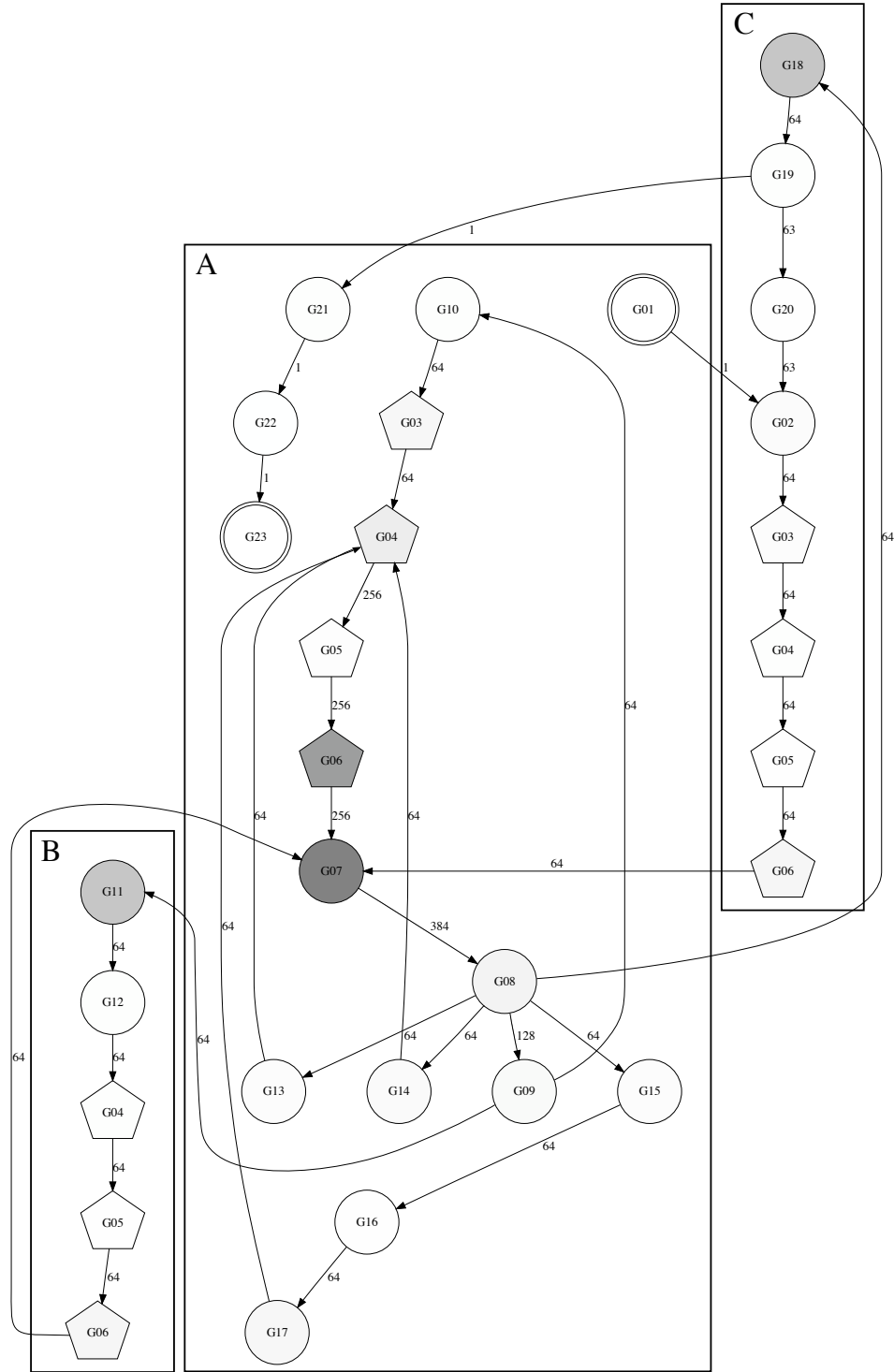
Figure 5: Software partition determined by the robust multiple instantiation model $\mathcal{AGRP}$. Nodes are shaded in proportion to their total inbound communication and computation time requirements at each location and arcs are labelled with the number of location aware control flows. Multiply instantiated nodes are shown as pentagons and singly instantiated nodes as circles. In contrast to Figure 4, the singly instantiated node G07 is the most time consuming node because of its high cross partition inbound communication costs.
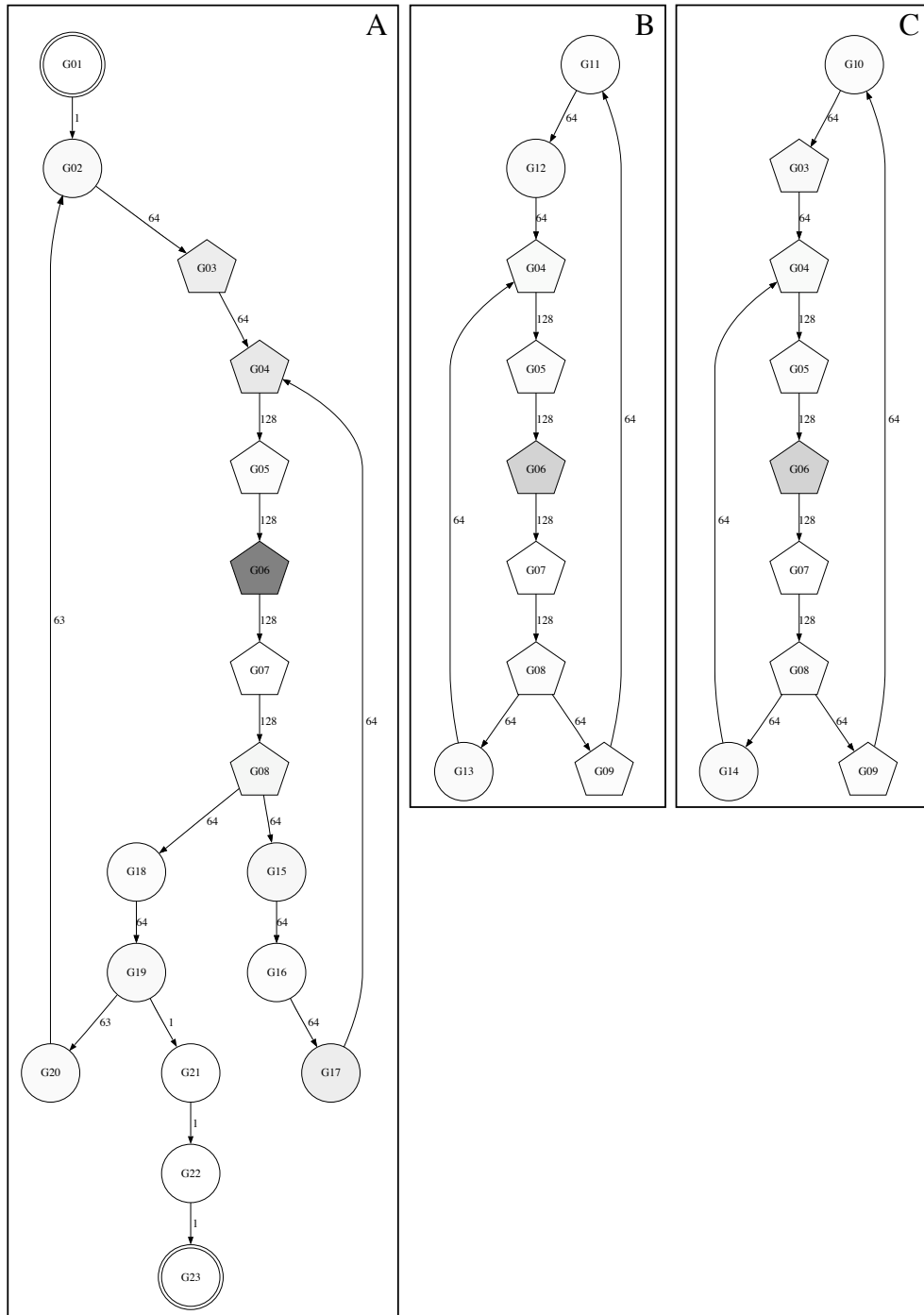
Figure 6: Software partition determined by the optimistic multiple instantiation model $\mathcal{OP}$. Note that the location-aware control flow is not connected. The shapes and shades of the nodes as well as the labels of the arcs have the same meaning as in Figure 5.
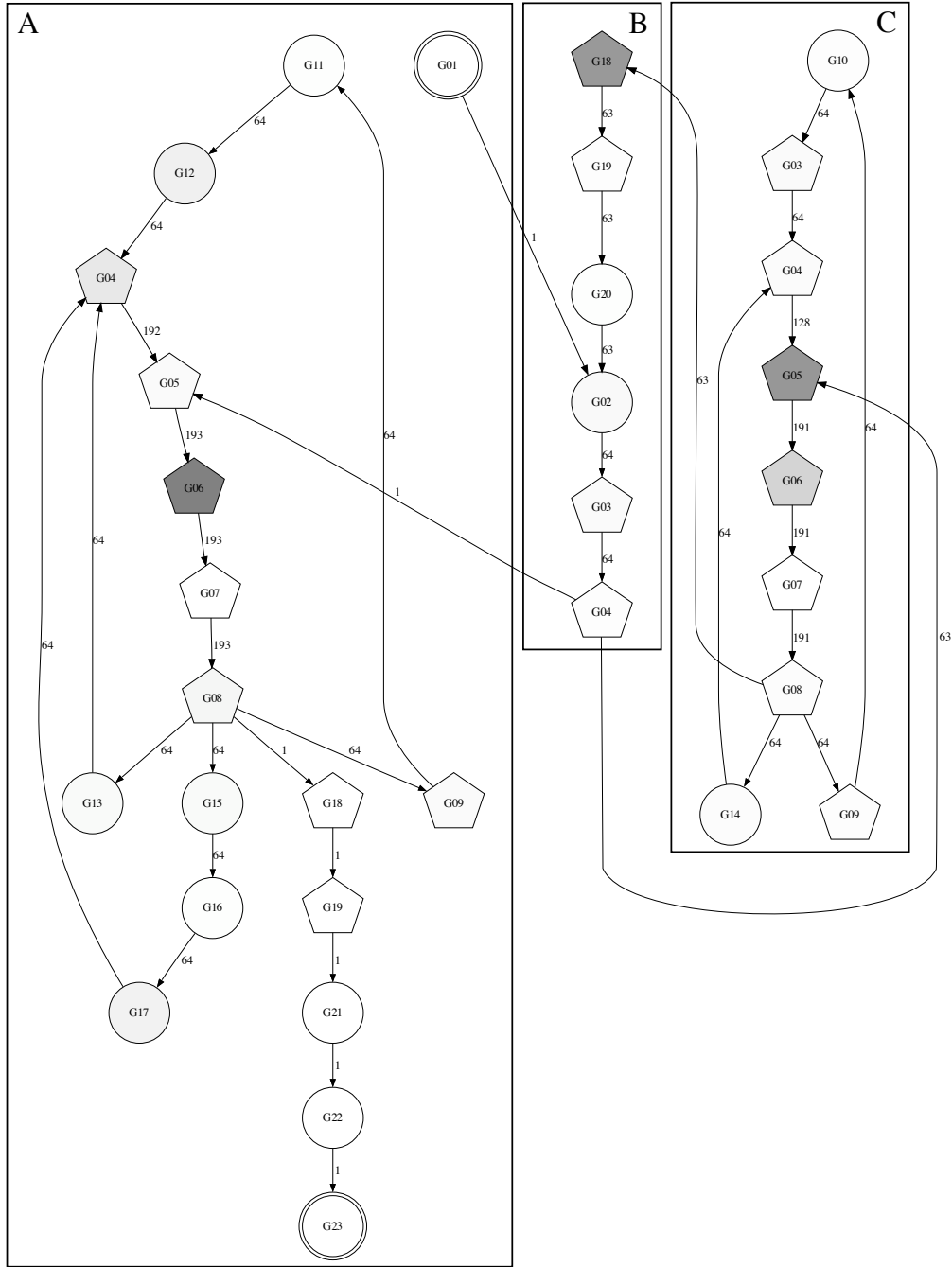
Figure 7: Software partition determined by $\mathcal{OP}^*$, the optimistic multiple instantiation model which enforces connectivity. Note in contrast to Figure 6, the location-aware control flow is now connected. The shapes and shades of the nodes as well as the labels of the arcs have the same meaning as in Figure 5.
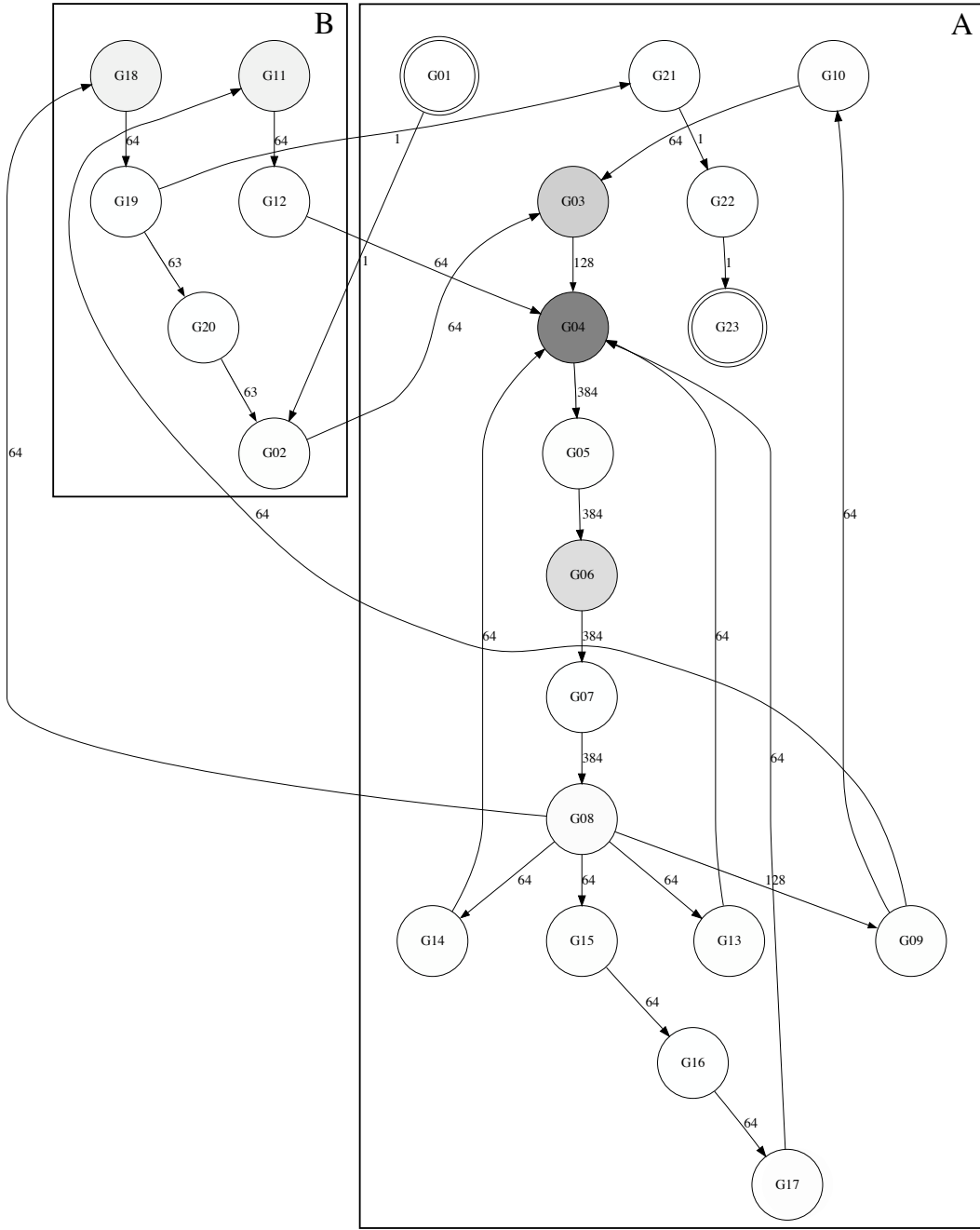
Figure 8: Software partition determined by the single instantiation model $\mathcal{P}_1$. Only two of the three execution locations were used. The shades of the nodes as well as the labels of the arcs have the same meaning as in Figure 5. All nodes are singly instantiated and are shown as circles.