A Decoupled Local-Memory Allocator

Boubacar Diouf, INRIA
Can Hantaş, Georgia Institute of Technology
Albert Cohen, INRIA and École Normale Supérieure de Paris
Özcan Özturk, Bilkent University
Jens Palsberg, UCLA

Compilers use software-controlled local memories to provide fast, predictable, and power efficient access to critical data. We show that the local-memory allocation for straight-line, or linearized programs is equivalent to a weighted interval-graph coloring problem. This problem is new when allowing a color interval to "wrap around," and we call it the submarine-building problem. This graph-theoretical decision problem differs slightly from the classical ship-building problem, and exhibits very interesting and unusual complexity properties. We demonstrate that the submarine-building problem is NP-complete, while it is solvable in linear time for not-so-proper interval graphs, an extension of the the class of proper interval graphs. We propose a clustering heuristic to approximate any interval graph into a not-so-proper interval graph, decoupling spill code generation from local memory assignment. We apply this heuristic to a large number of randomly generated interval graphs reproducing the statistical features of standard local memory allocation benchmarks, comparing with state-of-the-art heuristics.

Categories and Subject Descriptors: D.3.4 [**Programming languages**]: Processor — Compilers, Optimization

General Terms: Compiler, Algorithms, Performance

Additional Key Words and Phrases: Local memory, scratchpad memory, memory allocation, compiler

ACM Reference Format:
Diouf, B., Hantaş, C., Cohen, A., Palsberg, J., Özturk, Ö. 2012. A Decoupled Local Memory Allocator ACM Trans. Architec. Code Optim. V, N, Article A (January 2013), 23 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

Compilers use software-controlled local memories to provide fast, predictable, and power efficient access to critical data. Predictability to data access and power consumption efficiency are often essential to real-time and embedded applications. Most ARM processors have an on-chip local memory [ARM 1998], and more generally, it is typical for DSPs and embedded processors to have local memories, also called scratchpad memories [Motorola 1998; Instruments 1997]. More specialized processors also utilize local memories, including stream-processing architectures such as graphical processors (GPUs) and network processors [NVIDIA 2008; Burns et al. 2003]. Most processor(s) may directly access the *main memory* —typically off-chip DRAM— resources, but few exceptions exist. The IBM Cell broadband engine's synergistic processing units (SPU) [Kahle et al. 2005] which rely exclusively on DMA for instruction

This work is supported by the European Commission through the FP7 project TERAFLUX id. 249013.

and data transfers with main memory. Our approach is compatible with such memory models.

In systems with local memories, data transfers between the main memory and the local memory are inserted into the generated code by the compiler or the application. Previous studies addressed local memory management from different angles, targeting for both application/code and data. These efforts considered both static [Avissar et al. 2002; Sjödin and von Platen 2001; Steinke et al. 2002] and dynamic methods [Udayakumaran and Barua 2003; Li et al. 2009]. Static methods either place an array in the local memory or in the off-chip memory during the whole execution of a program. Dynamic methods place an array in the local memory at a certain moment and in the off-chip memory at a different moment depending on its access frequency. Dynamic methods take into account the dynamic behavior of the program.

In this paper, we consider an approach to local memory management that decouples spill code generation from local memory assignment. We show that the local memory allocation for straight-line programs or linearized programs, where the live ranges of variables or arrays are represented as intervals, is equivalent to a weighted interval-graph coloring problem that we call the submarine-building problem. The submarine-building problem differs slightly from the classical ship-building problem [Golumbic 2004] by allowing a color interval to "wrap around." We show that the submarine-building problem is NP-complete, while it is solvable in linear time for not-so-proper interval graphs, an extension of the the class of proper interval graphs. We propose a novel approach to approximate any interval graph into a not-so-proper interval graph, decoupling spill code generation from local memory assignment. We apply this heuristic to a large number of randomly generated weighted interval graphs reproducing the statistical features of standard local memory-allocation benchmarks. We compare our approach with state-of-the-art heuristics.

2. MOTIVATION

In a previous paper [Diouf et al. 2009], based on recent progress in register allocation, we considered a decoupled approach to local memory allocation, and we experimentally validated this decoupling. This paper takes a more theoretical stand point and seeks to better understand the optimization problem of local memory management.

Recent research in register allocation leverage the complexity and performance benefits of decoupling its allocation and assignment phases [Appel and George 2001; Pereira and Palsberg 2005; Hack et al. 2005; Bouchez et al. 2006b; Brisk et al. 2006]. The allocation phase decides which variables to spill and which to assign to registers. The assignment phase chooses which variable to assign to which register.

The allocation phase relies on the maximal *number* of simultaneously living sub-variables, called MAXLIVE, a measure of *register pressure*. When enough live-range splitting is done, it is sufficient that MAXLIVE is less or equal to the number of available registers to guarantee that all the sub-variables will be allocated and the forthcoming assignment phase can be done without further spill. In many cases, assignment can even be achieved in linear time [Hack et al. 2006; Bouchez et al. 2006b]. If at some program point the pressure exceeds the number of available registers, MAXLIVE needs to be reduced through spilling.

This decoupled approach permits to focus on the hard problem, namely the spilling decisions. It also improves the understanding of the interplay between live-range splitting and the expressiveness and complexity of register allocation. This is best illustrated by the success of SSA-based allocation [Bouchez et al. 2006a; Hack et al. 2006; Bouchez et al. 2006b; Bouchez et al. 2007; Braun and Hack 2009].

The intuition for decoupled register allocation derives from the observation that live-range splitting is almost always profitable if it allows to reduce the number of register spills, even at the cost of extra register moves. The decoupled approach focuses on spill minimization only, pushing the minimization of register moves to a later register coalescing phase [Appel and George 2001; Hack et al. 2006; Bouchez et al. 2008]. Here again, SSA-based techniques have won the game. Specifically, they collapse the register coalescing with the hard problem of getting out of SSA [Hack et al. 2006; Boissinot et al. 2009; Pereira and Palsberg 2009], as one of the last backend compiler passes.

The domain of local memory management tells a very different story. Some heuristics exist [Udayakumaran and Barua 2003; Udayakumaran et al. 2006; Kandemir et al. 2001; Li et al. 2009] but little is known about the optimization problem, its complexity and the interplay with other optimizations. The burning hot question is of course: does the decoupled approach hold for the local memory management problem? Surprisingly, the state-of-the-art of local memory management completely underexploit all the advances in register allocation. When focusing on arrays, the similarity between register and local memory allocation is obvious nonetheless:

Local memory allocation. Deciding which array blocks to spill to main memory and which array blocks to allocate to the local memory. Spilling is typically supported by DMA units.

Local memory assignment. Deciding at which local memory offset to assign which allocated array block.

In the context of local memory management, the maximum size of simultaneously living arrays[1], called `MaxSize`, gives a measure of *local-memory pressure*. Again, like for register allocation, live-range splitting helps to reduce the local-memory pressure. Since arrays are frequently accessed inside loops, local memory management algorithms often split arrays at loop-entry points , we call these points: decision points. Decision points can also be chosen in a finer manner, after loops or before array accesses. Local-memory pressure can also be reduced by loop-transformations like strip-mining, tiling which reduce the portion of accessed arrays. For all these reasons, the study of a decoupled approach in the local memory management context seems very appealing.

3. FROM LOCAL MEMORY MANAGEMENT TO WEIGHTED GRAPH COLORING

This section sets the terminology and definitions used in the rest of the paper.

3.1. Weighted Graphs

A graph $G = (V, E)$ consists of two sets, V the set of vertices, and E the set of edges. Every edge (v_1, v_2) of E has two end points $v_1 \in V$ and $v_2 \in V$. We consider undirected graphs only, i.e., we do not make difference between the edges (v_1, v_2) and (v_2, v_1).

A graph G is called an interval graph if its vertices can be put into one-to-one correspondence with a set of intervals I of a linearly ordered set such that two vertices are connected by an edge of G if and only if their corresponding intervals have a nonempty intersection.

Assuming each vertex v of $G = (V, E)$ is associated with a non-negative number $w(v)$, the weight of a subset $S \subset V$ is expressed as:

$$w(S) = \sum_{v \in S} w(v).$$

The graph G associated with the function w is called a weighted graph and denoted G_w. Moreover, G_w is a weighted interval graph if G is an interval graph.

[1]Not the number of simultaneously living arrays.

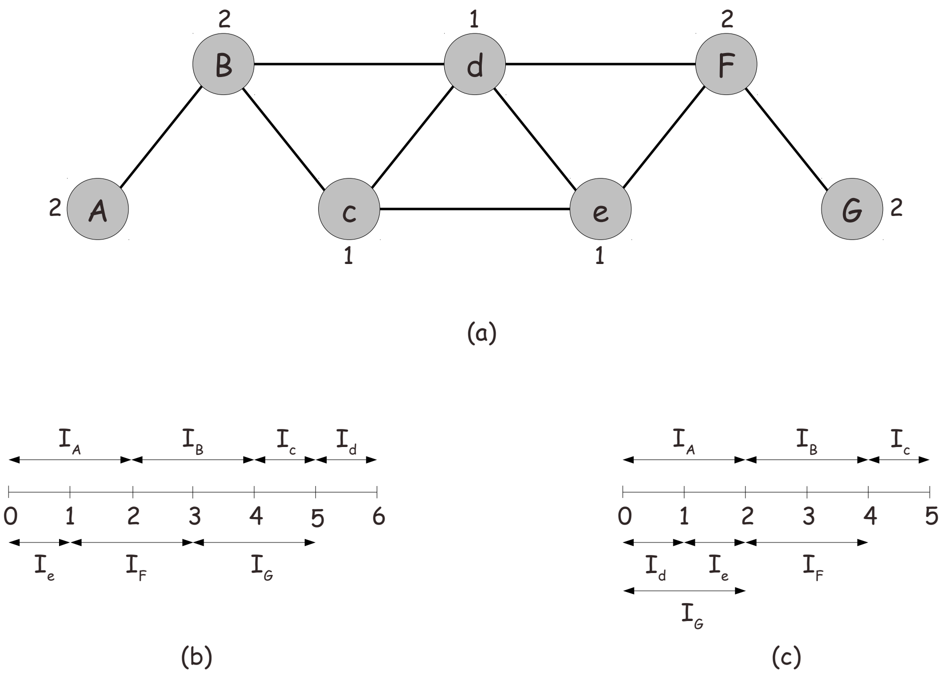

Fig. 1. Two colorings of a weighted graph.

An interval coloring of a weighted graph G_w is a function I mapping each vertex $v \in V$ onto an (open) interval I_v of $w(v) + 1$ consecutive integers of the real line, such that adjacent vertices are mapped to disjoint intervals; that is, $(v_1, v_2) \in E$ implies $I_{v_1} \bigcap I_{v_2} = \emptyset$. We say that I is a k-coloring of G_w if $I_v \in \{0, \ldots, k\}, \forall v \in V$. The chromatic number $\chi(G_w)$ is the smallest k for which we can find a k-coloring of G_w.

Figure 1(b) and Figure 1(c) show two colorings of the weighted graph shown in Figure 1(a). Figure 1(b) presents a 6-coloring of the weighted graph and Figure 1(c) shows a 5-coloring of the weighted graph. The chromatic number of this graph is 5.

3.2. Straight-Line Programs And Linearized Programs

Given an intermediate representation of an arbitrary program, the intermediate representation pseudo-instructions can be numbered according to some order. We define a *linearized program* as a program for which such kind of numbering has been performed and for each variable v in this program, we represent its live range as the live interval $[i, j[$, i being the number of the first instruction where v is first defined and j being the number of the instruction where v is last used. There can be some pseudo-instructions between i and j where v is not live, but with a successful live-range splitting this problem can become marginal. In the context of just-in-time compilation where compilation is critical, linearizing programs can pay off because it is fast to linearize a program and hopefully the produced code could be of relative good quality [Sarkar and Barik 2007].

3.3. Two Sides Of The Coin

We demonstrate the equivalence between allocating the local memory for a linearized program and coloring a weighted interval graph.

From linearized programs to weighted interval graphs. From a linearized program we construct a corresponding weighted graph called *interference graph*. For each variable in this program, we create a vertex and associate the size of the variable to this vertex. We create an edge between two vertices if there is a point in the program where the two variables are simultaneously live. Thus, an edge connects a pair of vertices if and only if the variables are simultaneously live. The constructed weighted graph is a weighted interval graph because each vertex corresponds to an interval defined by the definition point and the end point of the variable.

From weighted interval graphs to linearized programs. We use a method similar to the one presented by Lee et al. [Lee et al. 2008] to show that, for any weighted interval graph, we can exhibit a corresponding linearized program.

Chen [Chen 1992] and Saha [Saha et al. 2007] et al. have shown how to convert an interval graph with q intervals to an isomorphic *program like* interval graph in $\mathcal{O}(q\ log\ q)$ time. An interval graph is program-like if the intervals representing the vertices of the graph have start points and end points that are all different, and the start points and end points of the intervals form a set $\{1, \ldots, 2q\}$, where q is the number of intervals.

From a program-like weighted interval graph G_w, we construct in $\mathcal{O}(q)$ time the following straight-line program (with pseudo-C syntax) which consists of a set of $2q$ statements:

$$\forall i \in \{1, \ldots, 2q\} \begin{cases} \texttt{type}_I\ \texttt{v}_I\ \texttt{=}\ \cdots, \\ \text{where } \texttt{sizeof(type}_I\texttt{)} = w(I), \\ \text{if the interval } I \text{ of weight } w(I) \text{ begins at } i \\ \qquad\qquad \cdots\ \texttt{=}\ \texttt{v}_I, \\ \text{if the interval } I \text{ ends at } i. \end{cases}$$

4. WEIGHTED GRAPH COLORING

Thirty years ago, in her seminal paper [Fabri 1979], Fabri already envisaged to model the so-called problem of “automatic storage allocation” as a weighted graph coloring problem. She mentions the investigation of special subclasses of weighted graphs that are likely to occur. We construct a weighted graph G_w from a given linearized program. Finding an allocation for variables of the linearized program within a local memory of size k corresponds to finding a k-coloring of G_w.

This section introduces the *ship-building* problem which is related to weighted interval graph coloring. It also defines a new variant of the ship-building problem, called the *submarine-building* problem, very well suited to the local memory allocation problems on modern processors, and exhibiting interesting complexity results and approximation heuristics.

4.1. The Ship-Building Problem

We report here the ship-building problem as presented in the book of Golumbic [Golumbic 2004]. In certain shipyards the sections of a ship are constructed on a dry dock, called the welding plane, according to a rigid time schedule. Each section s requires a certain width $w(s)$ on the dock during construction. Can the sections be assigned space on a welding plane of total width k so that no spot is reserved for two sections at the same time?

Let the sections be represented by the vertices of a graph G and connect two vertices if their corresponding sections have intersecting time intervals. Thus G_w is a weighted interval graph. An interval coloring of G_w will provide the assignment of the sections to spaces, of appropriate size, on the welding plane. This assignment will be consistent with the intersecting time restrictions. The reader must be careful to distinguish between the time intervals which produced the edges of G_w and the color intervals which provide a solution to the assignment of space on the dock.

A weighted graph built from a linearized program associated with a number k (corresponding to the size of the local memory) is an instance of the ship-building problem. For a graph G_w and a number k, we call *ship*(G_w, k) an instance of the ship-building problem.

Determining whether $\chi(G_w) \leq k$ is an NP-complete problem [Golumbic 2004; Lee et al. 2008][2], even if G is an interval graph and the weight function w is restricted to the values 1 and 2. It follows that the ship-building problem is also NP-complete.

4.2. The Submarine-Building Problem

Since the local memory size is generally power-of-two, it is common to mask the addresses (in software or hardware) to let loads and stores wrap around to the local memory transparently. The submarine-building problem is a new variant of the ship-building problem. Like in the ship-building problem, a vertex must occupy a contiguous color-interval, but a circular allocation scheme can be adopted permitting to a color-interval to wrap around. It extends the ship-building problem's interval coloring to circular interval coloring. It follows that a solution of the ship-building problem is a solution to the submarine-building problem, but the converse is not generally true. Figure 2 shows an example of submarine coloring for the weighted graph in Figure 1(a). The inner circle represents the colors and each circular arc I_v represents a color interval assigned to the vertex v. The color interval I_F wraps around.

For a weighted graph G_w and a number k, we call *submarine*(G_w, k) an instance of the submarine-building problem. For the rest of the paper we say that G_w is k-ship-colorable, if *ship*(G, k) has a solution, and we also say that G_w is k-submarine-colorable, if *submarine*(G_w, k) has a solution.

To the best of our knowledge, this variant of the ship-building problem has never been carefully studied, and it has not been applied to the decoupling of the spilling and assignment problems in local memory management. Many open questions about fragmentation, optimality, complexity and feasibility are tied to this new variant of the ship-building problem.

Unfortunately, the submarine-building problem is also NP-complete on weighted interval graphs as we demonstrate below.

THEOREM 4.1. *The submarine-building problem is NP-complete.*

Proof. To show that the submarine-building problem is NP-complete, we first show that it is a problem in NP, and we then show how to build from an instance *ship*(G, k) of the ship-building problem an instance *submarine*$(G_w, k+1)$ of the submarine-building problem.

[2]This has been previously proved by Stockmeyer, but to the best of our knowledge, the proof of Lee et al. is the first publicly available one [Lee et al. 2008].

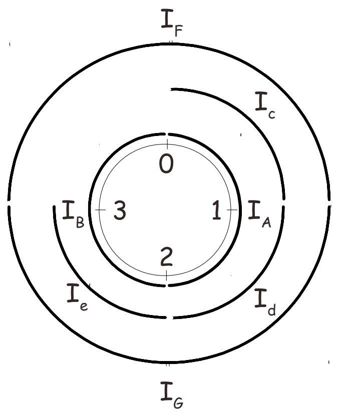

Fig. 2. An example of a 4-submarine-coloring.

A problem in NP. The submarine-building problem is in *NP* because a solution of the problem can be verified polynomially.

Reduction. From an instance *ship*(G_w, k) of the ship-building problem, we build an instance *submarine*$(G_w, k+1)$ of the submarine-building problem. Let f and ℓ be respectively the minimum of the start points of all intervals in G_w and the maximum of the end points of all intervals in G_w. The graph G'_w consists of all intervals of G_w and the interval β: $[f, \ell[$ of weight one.

Let θ be a solution of *ship*(G_w, k); θ maps each interval of G_w to a color interval between 0 and k. We define θ', a function mapping each interval α of G'_w to a color interval between 0 and $k+1$:

$$\forall \alpha \in G'_w \begin{cases} \theta'(\alpha) = \theta(\alpha) & \text{if } \alpha \in G_w \\ \theta'(\alpha) = [k, k+1[& \text{if } \alpha \notin G_w \end{cases}$$

It follows that θ' is a solution of *submarine*$(G_w, k+1)$.

Now, we study the converse case. Let θ' be a solution of *submarine*$(G_w, k+1)$. We define for a color interval $[s, e[$, an integer k, and the functions δ and mod:

$$\begin{aligned} \delta([s,e[, d) &= [s+d, e+d[\\ \mathrm{mod}([s,e[) &= [s \bmod (k+1), e \bmod (k+1)[\end{aligned}$$

Let $\theta'(\beta) = [s, s+1[$ (β is of weight one). We define for each interval α of G_w the function θ:

$$\theta(\alpha) = \mathrm{mod}\,(\delta(\theta'(\alpha), k-s))$$

The interval β of G'_w is live from f to ℓ, therefore there is no other interval of G'_w that occupies the color interval $[s, s+1[$. $\theta([s, s+1[) = [k, k+1[$, and the value on which function θ is equal to $[k, k+1[$ is $[s, s+1[$. Thus, θ assigns to each interval α of G_w an interval of some color between 0 and k. If two interfering intervals α and α' have two non-overlapping color intervals c and c' then $\theta(\alpha)$ and $\theta(\alpha')$ are non-overlapping too. It follows that θ is a solution of *ship*(G_w, k) if θ' is a solution of the *submarine*$(G_w, k+1)$.

5. WEIGHTED PROPER INTERVAL GRAPH COLORING

We study the properties of weighted proper interval graphs, a subclass of weighted interval graphs. This class is interesting because we will show the submarine-building problem is solvable in linear time for this class: any instance G_w of this class is colorable with $\omega(G_w)$ colors and in linear time. For this subclass, we also have a sufficient criterion permitting to decouple the ship-building problem.

5.1. Proper Interval Graph

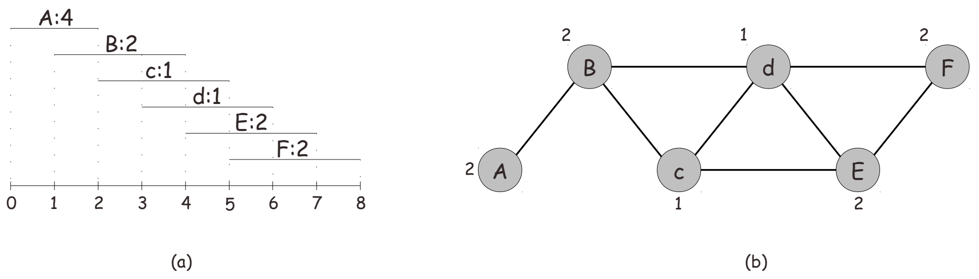

Fig. 3. An example of a weighted proper interval graph.

An interval graph G is a proper interval graph if it is constructed from a family of intervals such that no interval properly contains another [Golumbic 2004]. An interval graph is a unit interval graph if all of its intervals have the same length. It has been shown that the classes of proper interval graphs and the unit interval graphs coincide. A weighted graph G_w is a weighted proper interval graph if G is a proper interval graph. Figure 3 shows properly ordered weighted-intervals of the real line and their corresponding weighted proper interval graph.

5.2. Proper Ordering

Let us consider the representation of G_w, a weighted proper interval graph, on the real line, where the vertices of G_w correspond to intervals on the real line. Let us sort these intervals according to their start points. If two intervals i and i' start at the same point, we can place either i before i' or i' before i. This kind of ordering can be found for any weighted proper interval graph, and is called *proper ordering* in our approach. A proper ordering of the graph in Figure 3 is: A, B, c, d, E, F. Based on this ordering, we say that $i \prec i'$, if i is before i'.

LEMMA 5.1. *If $i \prec i'$ then, either i ends before i' or i and i' start and finish at the same time.*

Proof. $i \prec i'$ implies that either i starts before i' or i andi' starts at the same time:

— i starts before i'. Since i cannot properly contain i', then i ends before i'.
— i and i' start at the same time. Since, none of these two intervals cannot properly contain the other, then they end at the same time. □

5.3. Decoupled Submarine-Building Problem

Algorithm 1 performs a k-submarine-coloring of intervals of a weighted proper interval graph G_w. It takes as input a sequence of intervals of G_w sorted according to a proper ordering. It assigns to every interval a color interval contiguous to the color interval assigned to the previous interval (according to the proper ordering) in a clockwise manner. Finally, it gives a k-submarine-coloring of the graph as output.

Algorithm 1 SUBMARINEASSIGNMENTALGORITHM

```
Input: intervals: an array of properly ordered intervals
Var: $index \leftarrow 0$;
Var: map: an array associating to each interval an offset
1: for all $i \in$ intervals do
2:     map[$i$]$\leftarrow$ $index$ mod $k$;
3:     $index = index + weightOf(i)$;
4: end for
5: return map
```

THEOREM 5.2. *For any weighted proper interval graph G_w, Algorithm 1 guarantees a k-submarine-coloring if and only if $\omega(G_w) \leq k$.*

Proof.

Direct. k-submarine-coloring of $G_w \implies \omega(G_w) \leq k$.
Any k-submarine-coloring of G_w must assign color intervals that do not overlap to the intervals of a clique of weight $\omega(G_w)$ and this is only possible if $\omega(G_w) \leq k$.

Reciprocal. $\omega(G_w) \leq k \implies k$-submarine-coloring of G_w.
We will call a *point*, the moment an interval starts. Let P be the following property: "at point n, the live intervals $i_j, i_{j+1}, \ldots, i_n$ (sorted according to the proper ordering) are assigned to contiguous color intervals that do not overlap in a clockwise manner, in this order: *color*(i_j), *color*(i_{j+1}), $\ldots$ *color*(i_n)". The property P is an invariant at every point of Algorithm 1. If the graph contains m nodes, we have consequently m intervals and m points. The proof will be done inductively on points.
Just before the point 1, where the first interval i_1 starts, none of the color intervals are used. At point 1, algorithm 1 assigns to i_1 a color interval starting at 0 and property P is trivially satisfied.
Suppose that property P is satisfied from point 1 to point n, and let us see if property P is satisfied at point $n+1$ (we assume that we have at least $n+1$ intervals in the graph). We call d the number of dead intervals between n and $n+1$ (d can be zero, or $n-j$), we prove four claims successively:

(1) i_{j+d} is live. Indeed, if i_{j+d} was dead then all intervals preceding it would also be dead. Therefore, we would have $d+1$ intervals that are dead, which contradicts the definition of d.
(2) All the intervals between i_{j+d} and i_n are live too. If an interval i_k between i_{j+d} and i_n is dead then i_{j+d} is also dead because $i_{j+d} \prec i_k$; this leads to a contradiction with the first claim.
(3) From the two first claims and the satisfaction of proposition P at point n, we deduce that all live intervals $i_{j+d}, i_{j+d+1}, \ldots, i_n$ are assigned to contiguous colors that do not overlap, in a clockwise manner, in this order: *color*(i_{j+d}), *color*(i_{j+d+1}), $\ldots$ *color*(i_n).

(4) Algorithm 1 assigns to the new interval i_{n+1} a color interval contiguous to the last used color interval (the color of i_n) in a clockwise manner. Therefore, the color intervals assigned to live intervals are contiguous in a clockwise manner, in this order: *color*(i_{j+d}), *color*$(i_{j+d+1}), \dots$ *color*(i_n), *color*(i_{n+1}). The colors do not overlap because they are all contiguous and they do not exceed $\omega(G_w)$ which does not exceed k.

From the fourth claim, we conclude that at the point $n+1$ the property P is again verified.

Hence, using algorithm 1 guarantees that at every point, all the live intervals are assigned to contiguous color intervals that do not overlap, and the next starting interval will be assigned to a color interval. Thus, a k-submarine-coloring can be found for G_w if $\omega(G_w) \le k$.

As far as we are aware, this is the first time a decision problem is shown to be NP-complete on interval graphs and polynomial on unit interval graphs (which are equivalent to proper interval graphs).

6. WEIGHTED NOT-SO-PROPER INTERVAL GRAPHS

We say that two intervals A and B *properly interfere* if A interferes with B such that A strictly starts before B and B strictly ends after A or vice versa.

We define a weighted Not-So-Proper (NSP) interval graph as a weighted interval graph, where each pair of properly interfering intervals A and B, is such that A and B must not be contained in any other interval of the graph.

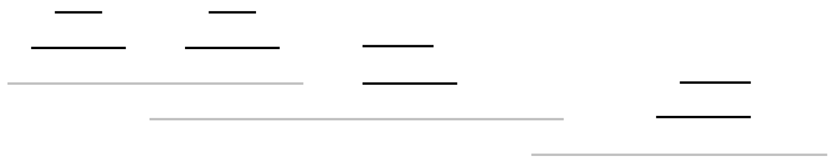

Fig. 4. An example of weighted NSP graph.

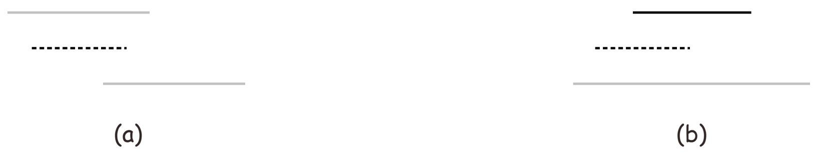

Fig. 5. Two graphs that are not weighted NSP graphs.

Figure 4 shows an example of a weighted NSP graph (weights have been omitted in the figure), whereas Figure 5 illustrates two weighted graphs that are not NSP. The light gray lines represent intervals that are not contained in other intervals, the solid black lines represent intervals that are contained, and the black dashed lines represent the intervals we do not want to have in weighted NSP graphs.

The weighted NSP graphs are the class of graphs that includes the weighted proper interval graphs and the superperfect graphs defined by Li et al. [Li et al. 2011]. Thus, when the submarine assignment problem is considered, the weighted NSP interval graphs are guaranteed to be `MaxSize`-colorable.

Algorithm 2 NSPASSIGNMENTALGORITHM

```
Input: intervals: a list of intervals sorted by increasing start point
Var: map: an array associating to each interval an offset
Var: stack: a stack used to keep track of contained intervals
 1: offset ← 0
 2: container ← ⊥
 3: for all $i \in intervals$ do
 4:   if container $= \perp \vee \neg$(CONTAINS($container, i$)) then
 5:     container ← $i$
 6:   end if
 7:   while stack $\neq \emptyset$ do
 8:     if $\neg$(CONTAINS(PEEK(stack), $i$) then
 9:       contained ← POP(stack)
10:       offset ← (offset + MAXSIZE − WEIGHTOF(contained)) mod MAXSIZE
11:     else
12:       break out of the loop
13:     end if
14:   end while
15:   if container $\neq i \wedge$ (CONTAINS(container, $i$)) then
16:     PUSH($stack, i$)
17:   end if
18:   map[$i$]← index mod k
19:   $index = index +$ WEIGHTOF($i$)
20: end for
21: return map
```

Algorithm 2 performs a submarine assignment for a weighted NSP graph on a local memory of size `MaxSize`. It receives as input $intervals$, a list of intervals sorted by increasing start point, and returns at the end map, a map that associates to each interval an offset into the local memory. This algorithm differentiates between intervals that are not contained in any other interval, called *containers* and those contained in an interval. The contained intervals are stocked into $stack$. The variable $container$ keeps track of the last starting container. When a new interval, i, starts, it is verified for containment — the function CONTAINS($container, i$) returns true, if i is contained in $container$ and false otherwise — and if it is not contained in the currently live intervals, it is set as the new container. Then, all the dead intervals are removed from $stack$ and the offset is updated. The function WEIGHTOF(i) returns the weight of the interval i. If i is contained into another interval, it is pushed onto $stack$. Finally, i is assigned to the current offset, which is then updated.

7. LOCAL MEMORY ALLOCATION THROUGH WEIGHTED NSP GRAPH COLORING

As explained in the section 3, from a linearized program it is possible to construct a corresponding weighted interval graph. If the resulting graph is a weighted NSP interval graph, when the submarine assignment problem is considered, it is always possible to use `MaxSize` as a criterion to ensure that the assignment phase is feasible without spills. Thus, the allocation algorithm can be decoupled thanks to the `MaxSize` criterion. For arbitrary interval graphs, the problem is NP-complete and a heuristic-based solution should be envisaged.

We devised a solution that takes advantage of our submarine assignment algorithm. This solution that decouples the allocation and assignment, performs the two following steps:

(1) it approximates an arbitrary weighted interval graph into a weighted NSP graph through spilling and splitting.
(2) it performs a submarine assignment with Algorithm 2.

7.1. Live Ranges Representation

Each array of a program is represented by a live interval (*compound interval*) starting from its first definition to its last use. Within the compound interval, it may exist some idle sub-intervals where the array is not accessed at all and other sub-intervals where the array is frequently accessed. The latter are called *basic intervals*. These basic intervals often correspond to loops, because arrays are generally frequently accessed through loops. Figure 6 shows an example of compound interval (ci) and and the basic intervals (bi_1, bi_2, bi_3, bi_4) that compose ci. Usually it is not recommended to split the live interval of an array inside a loop, because the incurred memory transfer cost will be too high, hence in our approach we do not split a compound interval inside one of its basic interval.

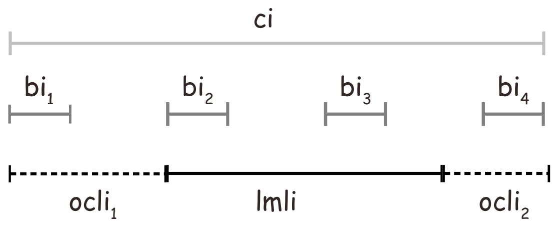

Fig. 6. An example of live intervals.

During the execution of a program, an array may occupy different locations in memory. The sub-live range — a sub-interval of the array's compound interval — during which the array occupies a specific location in memory, is called a *memory live interval*. A memory live interval is composed of a set of successive basic intervals of the compound interval. All the basic intervals in a memory live interval belong to the same compound interval. But, on the other hand, two basic intervals of the same compound interval may belong to different memory live intervals. If the array occupies a unique location in memory during all its compound interval, the memory live interval is the same as the compound interval. If the array is located in the local memory during the memory live interval, it is called a *local-memory live interval*. In contrast, if it is in the off-chip memory, it is called *off-chip live interval*. Figure 6 shows an example of configuration that can happen during execution. The array A which has ci as compound interval is in the local memory during the interval $lmli$, a local memory live interval. A is in the off-chip memory during $ocli_1$ and $ocli_2$, two off-chip live intervals.

7.2. The approximation algorithm

```
Algorithm 3 APPROXIMATE
Input: basics: an array containing all the basic intervals of the program sorted by
    increasing start point
Input: lm_size: the size of the local memory
Var: active: a list that keeps track of local-memory live interval currently live
Var: map: an array associating to each compound interval its current mem-li
Var: loc-mem-lis: a list containing all the local-memory live intervals
1: for all $bi \in basics$ do
2:     EXPIREOLDLOCALMEMORYLIVEINTERVALS($active, bi.start$)
3:     FINDMEM-LI($bi, active, map, loc\text{-}mem\text{-}lis, lm\_size$)
4: end for
5: return loc-mem-lis
```

Based on the notions of local-memory live interval and off-chip live interval, we devise an approximation algorithm which transforms a weighted interval graph into a weighted NSP interval graph composed of local-memory live intervals through spilling and splitting of the live intervals. This approximation is performed by Algorithm 3.

Algorithm 3 receives as input *basics*, an array containing all the basic intervals of the program sorted by increasing start point, and *lm_size* the size of the local memory. It then creates two variables, *active* and *map*. At every moment, *active* keeps track of the local-memory live intervals that are live and *map* associates to each compound interval its latest computed memory live interval. For each basic interval, *bi* , Algorithm 3 first removes from *active* the local-memory live intervals of a compound interval whose end point is lower than the start point of *bi*. Afterwards, Algorithm 3 attempts to add *bi* into a local-memory live interval. If this is impossible *bi* becomes part of an off-chip live interval. When all of the basic intervals of the program are processed, Algorithm 3 returns *loc-mem-lis*, the list of all the local-memory live intervals which are the intervals of the approximated weighted NSP interval graph.

The weighted NSP interval graph is computed iteratively with Algorithm 4. Algorithm 4 aims to assign *bi*'s array to a location in the local memory based on the currently active local-memory live interval. It first retrieves the latest memory live interval of *bi*'s compound interval (*bi.compound*) into *mem-li*. Three different cases can happen:

(1) *mem-li* does not correspond to any value. It means that this is the first time the array of the compound interval is accessed. In this case Algorithm 4 creates *new_mem-li*, a temporary new local-memory live interval with *bi*, and checks, with the function ALLOCATEORNOT(), if it can be added to *active*. If it is the case, *new_mem-li* is added to *active* and to *loc-mem-lis*, and *bi*'s compound is is associated to *new_mem-li*. Otherwise, *bi* will be part of *off-chip_mem-li*, an off-chip live interval, and *bi*'s compound is associated to *off-chip_mem-li*.
(2) *mem-li* corresponds to an off-chip live interval. If the cost of loading *bi*'s array is lower than the cost of accessing it from the off-chip memory — the function ISBENEFICIALTOLOAD() returns true— and there is enough room for it — the function ALLOCATEORNOT() returns true — then *mem-li*'s end point is set to *bi*'s start point, *new_mem-li* is added to *active* and to *loc-mem-lis*, and *bi*'s compound is associated to *new_mem-li*. Otherwise, *bi* will be part of *mem-li*, which is an off-chip live interval in this case.

Algorithm 4 FINDMEM-LI

```
Input: bi: the starting basic interval
Input: active: a list that keeps track of local-memory live interval currently live
Input: map: an array associating to each compound interval its current memory live
    interval
Input: loc-mem-lis: a list containing all the local-memory live intervals
Input: lm_size: the size of the local memory
 1: mem-li ← map[bi.compound]
 2: // If it is the first time when the compound of bi appears
 3: if mem-li = ⊥ then
 4:   new_mem-li ← CreateNewloc-mem-li(bi)
 5:   if AllocateOrNot(new_mem-li, active, lm_size, loc-mem-lis, map) then
 6:     Add new_mem-li to active
 7:     Add new_mem-li to loc-mem-lis
 8:     map[bi.compound] ← new_mem-li
 9:   else
10:     off-chip_mem-li ← Spill(bi.start, new_mem-li, active)
11:     map[bi.compound] ← off-chip_mem-li
12:   end if
13:   // If bi.compound was previously in the off-chip memory
14: else if mem-li is a off-chip-li then
15:   new_mem-li ← CreateNewLocalMemoryLiveInterval(bi)
16:   if IsBeneficialToLoad(mem-li, bi)
      ∧ Accept(new_mem-li, active, lm_size, loc-mem-lis, map) then
17:     mem-li.end ← bi.start
18:     Add new_mem-li to active
19:     Add new_mem-li to loc-mem-lis
20:     map[bi.compound] ← new_mem-li
21:   else
22:     Mark that bi is also represented by mem-li
23:   end if
24:   // If bi.compound was previously in the local memory
25: else
26:   Mark that bi is also represented by mem-li
27: end if
```

(3) *mem-li* corresponds to a local-memory live interval. In this case, Algorithm4 marks that *bi* will be part of *mem-li*, which is a local-memory live interval in this case.

The function ALLOCATEORNOT() used in Algorithm 4 is described by Algorithm 5. This algorithm spills or splits either the active local-memory live intervals or spills *new_mem-li* which corresponds to transforming it into an off-chip live interval. When Algorithm 5 is invoked, it checks if the sum of the weights of *new_mem-li* and the local-memory live intervals currently in *active* — given by the function WEIGHTOF() — is lower than *lm_size*, the size of the local memory. In this case, it returns true to report that *new_mem-li* can be added to *active*. Otherwise, Algorithm 5 considers three possibilities:

(1) First, it computes the cost of splitting the local-memory live intervals when *new_mem-li* starts. Note that the local-memory live intervals, currently used at that moment, which have a basic interval that contains the start point of *new_mem-li*, could not be split at that moment. Algorithm 5 computes the cost of spilling

Algorithm 5 ALLOCATEORNOT

```
Input: $new\_mem\text{-}li$: the new memory live interval
Input: $active$: a list that keeps track of local-memory live intervals currently live
Input: $map$: an array associating to each compound interval its current memory live
    interval
Input: $loc\text{-}mem\text{-}lis$: a list containing all the local-memory live intervals
Input: $lr$: a local-memory live interval we want to add to $active$ if possible
Input: $lm\_size$: the size of the local memory
 1: if $\text{WEIGHTOF}(active) + new\_mem\text{-}li.weight \leq lm\_size$ then
 2:     return TRUE
 3: else
 4:     $splitting\_cost \leftarrow \text{COSTOFSPILLBEFORESPLIT}(lr.start, active)$
 5:     $spilling\_cost \leftarrow \text{COSTOFSPILL}(lr.start, active)$
 6:     $lr\_spill\_cost \leftarrow \text{COSTOFSINGLESPILL}(lr)$
 7:     $min \leftarrow \text{MINIMUM}(splitting\_cost, spilling\_cost, lr\_spill\_cost)$
 8:     if $min = splitting\_cost$ then
 9:         $\text{SPILLBEFORESPLIT}(lr.start, active, loc\text{-}mem\text{-}lis, map)$
10:     else if $min = spilling\_cost$ then
11:         $\text{SPILL}(lr.start, active, loc\text{-}mem\text{-}lis, map)$
12:     else
13:         $\text{SINGLESPILL}(lr, map)$
14:     end if
15: end if
```

these local-memory live intervals first, then if some space is still needed to hold *new_mem-li*, it computes the cost of breaking some local-memory live interval into a local-memory live interval which stops now, at the moment when *new_mem-li* starts, and an off-chip live interval from now. It breaks the local-memory live intervals until *new_mem-li* can fit.

(2) It computes the cost of spilling some local-memory live intervals until *new_mem-li* can fit. Algorithm 5 walks over the local-memory live intervals in the order of increasing spill cost.

(3) It computes the cost of spilling *new_mem-li*.

Algorithm 5 selects the possibility that is the cheapest and updates accordingly the variables *map*, *active*, and *loc-mem-lis*.

8. EXPERIMENTAL EVALUATION

To evaluate our approach we have generated 2000 graphs (1000 superperfect graphs and 1000 arbitrary interval graphs) with various properties. We compared our approach with the polynomial allocator of Li et al. [Li et al. 2009] which is the closest work to ours. We also compare our approach with the classical best-fit allocator and to a variant of the best-fit which aims to reduce copies. Notice both best-fit algorithms are exponential in the size of the local memory: they are very effective for a few hundreds of kilobytes, but they may not scale to larger allocation problems such as those arising on the local memories of GPU accelerators.

The results collected so far are very encouraging and indicate that the proposed approach competes with the best-fit, and may outperform it on the harder allocation problems with a small local memory size.

We start with the survey of our experimental methodology, then we provide an experimental evaluation of our heuristic and discuss the results.

Table I. Model parameters.

Constant	Latency
latency_local_memory	8
latency_main_memory	128
$latency\_move(s_v)$	$8 + 2s_v$
$latency\_spill(s_v)$	$128 + 4s_v$
$latency\_reload(s_v)$	$128 + 4s_v$

8.1. Methodology

Our comparison are based upon randomly generated graphs that attempt to reproduce the characteristics of real life applications. We were inspired by the common characteristics found in representative numerical benchmarks, such as those we used in our previous work [Diouf et al. 2009] (`BMCM` [Berry et al. 1988], `MXM`, `EDGE_DETECT` [Lee 1998] and `FFT` [Lee 1998]) and the Polybench [Pouchet 2012]. These graphs have been used to evaluate our decoupled approach to the local memory allocation problem.[3]

Table I lists the parameters that model the local memory used in our experiments. We modeled a typical DDR memory with burst/pipelined and random-access latencies. Varying these ratios will change the overall benefits of local memory allocation, but not the relative performance of the different heuristics/algorithms.

Before we present the results of our evaluation, we first explain how we proceed to generate the random graphs, then we give a description of the compared algorithms, and finally we depicts the details of our experimental evaluation.

8.1.1. Graph Generation. We generate interval graphs, selecting start (left) and end (right) points at random. We also try to reproduce the control and data structures found in real life applications, adding specific constraints and limits on these graphs. We control the number of intervals, the minimum length of an interval, the maximum number of concurrently live intervals and weight of intervals. Although there is no specific limit on maximum number of data structures that can be used in an application, in practice it is not usual to see hundreds of data structures in a single application. Also data structures in applications are usually not intended to be created and used at the same line of code. That is when a data is created it is meant to be used for some time, they are created, used and removed from memory when they are no longer required. By giving a bound on interval length we have ensured that the virtual data structures created within an application are used for a minimum length of time and the maximum number of live intervals within the graph is used to limit the maximum number of data elements concurrently used in an application.

Our graphs are composed of compound intervals which are in turn composed of basic intervals which have been introduced in Section 7. Each compound interval represents the whole live range of an array, and the basic intervals represent the sub-intervals of the compound interval where the array is frequently accessed. These basic intervals are called hot portions in the approach of Li et al. [Li et al. 2011]. Usually the array are accessed frequently through loops and in our implementation, each basic interval corresponds to an access to an array through a loop. A basic interval begins at the start point of a loop where an array is accessed and ends at the end point of that loop. Thus to generate the basic intervals, we first generate some loops that can be imbricated or not. Each loop has a start point and a end point that differs from start points and end points of other loops. A loop l_1, that contains a loop l_2, starts before l_2 and ends

[3]Extracting these graphs from real code would strengthen the experimental evaluation, assuming the extraction method implements loop transformations to improve data locality and to partition arrays into small, homogeneous blocks. Interference graphs may then be built from the live ranges of these array blocks [Diouf et al. 2009]. How to perform these steps automatically and profitably remains a largely open problem.

Table II. Parameters for graph generation.

Parameter	**Value**
maximal loop nest	4
number of loops	20
minimal number of_outerloops	5
maximal number of outerloops	5
number of compound intervals	30
maximal number of basic intervals per compound interval	3
The number of different array's size	10

after l_2. For each array A, we randomly choose some loops where A will be frequently accessed. A basic interval bi of A corresponds to a chosen loop l. The start point and the end point of bi are set to the start point and end point of l. When choosing the basic intervals of A, when two basic intervals are such that the one is contained in the other, we consider that the A is accessed in the containing basic interval. Each bi is associated to a randomly generated frequency and is defined as being a write, meaning that A is modified within bi , or a use, meaning that A is only read within bi. After all the basic intervals of A are chosen, we define ci, the compound interval of A: it contains all the basic intervals of A, starts with the ourtermost loop containing the first basic interval of A, and ends with the outermost loop containing the last basic interval of A.

To make the generation of interval graphs as generic as possible we use the parameters presented in Table II. We were inspired by the frequent characteristics found in representative numerical benchmarks, such as the 4 above mentioned kernels and the Polybench [Pouchet 2012]. The parameters allow to control the number of intervals in the graph, the number of loops, the maximum depth of a loop, the maximum number of basic intervals of a compound interval, etc.

8.1.2. Algorithms. We use the randomly generated graphs to evaluate our approach by comparing it with three other approaches:

Classical Best fit. This algorithm, denoted `BestFit`, walks over the list of basic intervals and attempts to assign every basic interval bi to a portion of the local memory(in facts, it is the array of this bi which is assigned). If there is enough space to hold bi, it chooses the space where bi fits the best. Otherwise either bi is spilled or some previously assigned basic intervals are spilled to make room for bi.
Best fit variant. This is a variant of of the best-fit, denoted `BestFitVariant`. Whenever a basic interval bi ends, it is checked if bi is the last basic interval of its compound interval ci. If so, bi is removed from the local memory. Otherwise, bi is left in the local memory until this space is needed for another basic interval bi' of another compound interval ci' (another array) or the next basic interval of ci starts. The aim of this technique is to assign, if possible, different live ranges of an array to the same offset in the local memory in order to avoid copy costs.
Superperfect. This is our implementation of the approach of Li et al [Li et al. 2011] denoted here `SuperPerfect`. In this implementation the live range splitting is naturally performed because the frequently accessed portions (called hot portions by authors of the approach) of array's live ranges are exactly the basic intervals. Thus, the set of candidates subject to allocation are composed of the compound intervals and their basic intervals. The allocation algorithm will consider either the compound interval or each of its individual basic intervals for allocation, but not both at the same time. To approximate a given graph into a superperfect graph, we go through the loops in their increasing start points, and we mark all the intervals (the candidates, the basic and compound intervals) defined at the same point as containing related, notice that two different loops have start points and end points that differ. When at a start point, an interval j starts when another interval i is

already defined and still live, and i and j are not containing related, we extend i to the end of j. After this approximation the new formed interval graph is superperfect.

8.1.3. Evaluation Details. Our evaluation was conducted on different local memory sizes. We varied the size of the local memory in accordance with `MaxSize` when performing the experiments; `MaxSize` — the maximum size of simultaneously living arrays — can vary significantly between two graphs. Thus, we do think that it is not very relevant to compare two allocation algorithms on graphs with `MaxSize` that widely differs when using a local memory with a fixed size. We focus here on the interesting case where `MaxSize` is the maximum size of simultaneously living *basic intervals* and not of the compound intervals, which is larger and thus make the problem easier to solve.

We based our comparison on the cumulative memory access latency incurred by each method. The allocation of the best quality is the allocation with the lowest access latency. For every interval graph the cost of the allocation performed by `BestFitVariant`, `SuperPerfect` and our approach, denoted `NSP`, have been normalized with respect to `BestFit`. For a given size of the local memory and a given algorithm, an associated bar shows the average (Figure 7 and Figure 9) and how the individual allocations are statistically distributed in the normalized allocation space (Figure 8 and Figure 10), of all the normalized allocation cost of the given algorithm.

To perform an allocation with our approach we feed Algorithm 3 with the list of basic intervals of each weighted interval graph. Algorithm 3 will then return a Not-So-proper interval graph composed of local-memory live intervals. The `MaxSize` of the resulting graph is less than or equal to the size of the local memory, hence all its associated array blocks can be placed in the local memory using Algorithm 2.

8.2. Results and Discussion

We provide here an evaluation of our approach, on 1000 of randomly generated superperfect graphs and 1000 of randomly generated arbitrary interval graphs. The results presented in this section show the performance of `BestFitVariant` variant (light gray bars), `SuperPerfect` allocator (gray bars) and our approach denoted `NSP` (black bars).

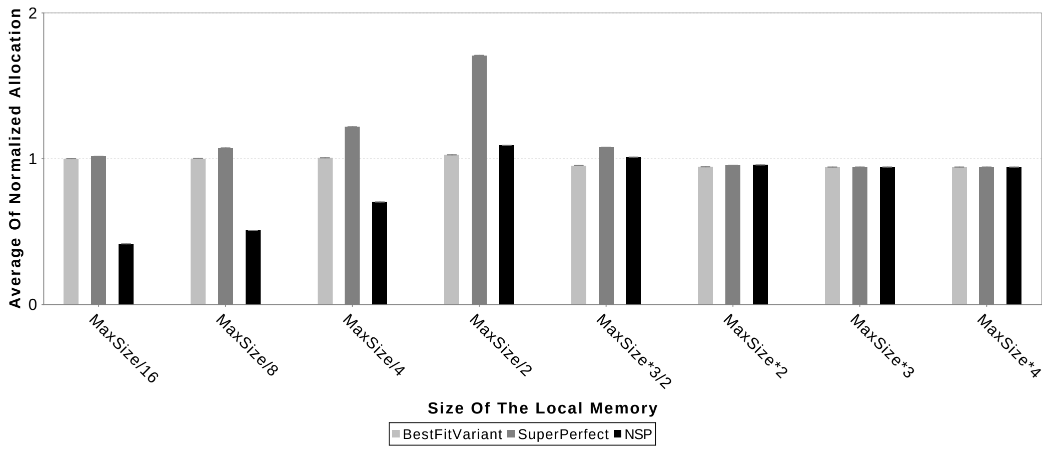

Fig. 7. Average of normalized allocation w.r.t. best fit on superperfect graphs.

Figure 7 and Figure 8 present the results obtained on superperfect graphs. For all local memory sizes, our approach is better than `SuperPerfect` approach. Since the

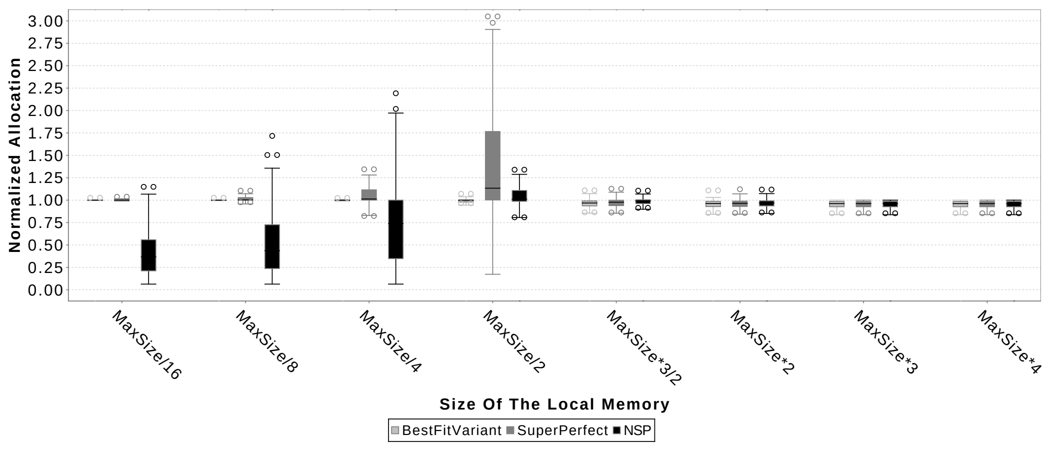

Fig. 8. Distribution of normalized allocation w.r.t. best fit on super perfect graphs.

graphs are superperfect, there is no need of approximation for both approaches. Thus, the difference here is on the choice of allocated arrays. This shows that, on the generated graphs, our approach performs a better allocation. When the size of the local memory is lower or equal to `MaxSize`$/4$, our approach is better than `BestFit` and `BestFitVariant`. For local memory size of `MaxSize`$/2$, `MaxSize`, `MaxSize` $\times\, 3/2$, `BestFit` and `BestFitVariant` gives better results. For a size of the local memory going from `MaxSize` $\times\, 2$ to `MaxSize` $\times\, 4$, `BestFitVariant`, `SuperPerfect` and `NSP` give similar results with small standard deviation. Algorithms produce approximately the same allocation because the size of the local memory is big enough to allow a good allocation. Since the three algorithms try to assign to the basic intervals, of a same array, the same place in the local memory, avoiding thus extra copies, they give better results compared to `BestFit`. But the small improvements suggest that these copies does not have a big impact on the global allocation cost.

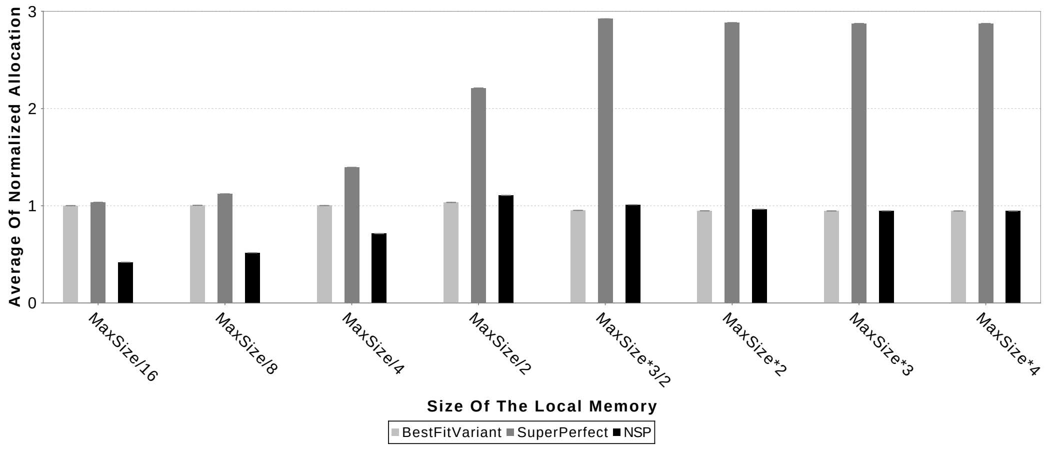

Fig. 9. Average of normalized allocation w.r.t. the best fit on arbitrary graphs.

Figure 9 and Figure 10 show the results obtained on arbitrary graphs. Here again, for all local memory sizes, our approach gives better results than `SuperPerfect` ap-

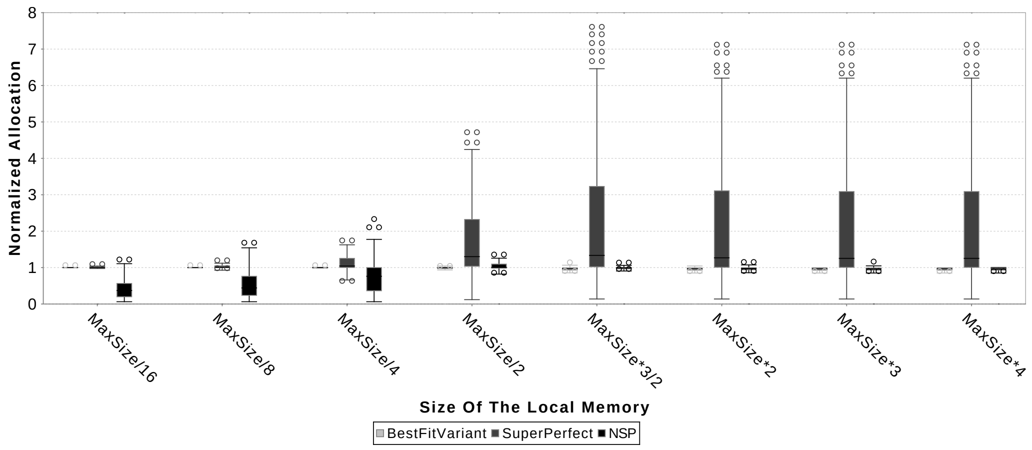

Fig. 10. Distribution of normalized allocation w.r.t. the best fit on arbitrary graphs.

proach, and on arbitrary graphs. Our approach, like `BestFit` and `BestFitVariant`, performs on average allocations twice better than the allocations performed by `SuperPerfect` algorithm. This performance degradation, of the `SuperPerfect` algorithm, is mostly due to its approximation algorithm which is not meant to be used on graphs that are not quasi-superperfect, that are interval graphs that have most of their intervals matching the containment property. Comparing with `BestFit` and `BestFitVariant`, our approach gives similar results to those presented in Figure 8, but with a slightly higher variability.

9. RELATED WORK

Strong links between register allocation and local memory management have been discovered for more than 30 years by [Fabri 1979]. Fabri's seminal paper also studied the interplay between local memory management and the loop transformations. Since then it has been ignored in the field of local memory management and register allocation [Appel and George 2001; Hack et al. 2006; Bouchez et al. 2006b; Quintáo Pereira and Palsberg 2008]. While previous studies addressed local memory management from different angles, targeting both code and data, we are especially interested in data management [Kandemir et al. 2001; Issenin et al. 2007; Dominguez et al. 2007]. We target dynamic methods which are superior to static ones except when code size is extremely constrained [Udayakumaran and Barua 2003]. We elaborate on two recent series of results targeting stack and global array management in local memories, embracing the analogies with register allocation. The first approach [Li et al. 2005] uses an existing graph coloring technique to perform memory allocation for arrays. It partitions the local memory for each array size, performs live range splitting and uses a register allocation framework to perform memory coloring. The second approach [Avissar et al. 2002; Udayakumaran and Barua 2003; Udayakumaran et al. 2006] allocates data onto the scratch-pad memory between program regions separated by specific program points. More specifically, allocation is based on the access frequency-per-byte of a variable in a region (collected from profile data). Program points are located at the beginning of a called procedure or before a loop entry.

The closest work to ours is [Li et al. 2011], where authors observed that in many embedded applications most arrays present a specific live range behavior. Specifically, for any two array, live ranges are either disjoint or one of the arrays is contained by the other one (containment property). They showed that, for the tested benchmarks, it is extremely rare to have two live ranges interfere with one another without containment. They extend the live range of one of the arrays to contain the other When this happens. Authors proved that the interference graph of an application with such a property is a comparability graph which is a superperfect graph and hence optimal interval-coloring for this array interference graphs is possible. Based on this observation, they rely on the maximum weighted clique to guarantee the optimal colorability of generated interference graph. When the maximum weighted clique exceeds the size of the local memory, they use heuristics to spill or split some of the live ranges. While this work is interesting, it is restricted to applications where most arrays satisfy the containment property. On the other hand, our work leverages the decoupled allocation/assignment approach, allowing scalable and more effective algorithms. Moreover, it offers much more flexibility in terms of integration of architecture constraints and performance models.

10. CONCLUSION

We implemented a novel compilation-time local memory management approach through decoupling spill code generation and local memory assignment. We represent the live range intervals of variables and arrays as a weighted interval graph. We defined a new decision problem called the submarine-building problem, a variant of the ship-building problem. The submarine-building problem corresponds to coloring a weighted interval graph with a cyclic set of colors (corresponding to a wrap-around local memory). We demonstrate important complexity results on this problem, some of which particularly original in graph theory. We provide a new clustering heuristic to approximate interval graphs into not-so-proper interval graphs, on which the submarine-building problem can be decided in linear time. This approximation effectively decouples the generation of spill code from the local memory assignment problem. Our preliminary experiments demonstrate the practicality of the approach, and very favorable allocation results compared to state-of-the-art allocators.

Acknowledgments. This work was partly supported by the European FP7 project TERAFLUX id. 249013.

REFERENCES

APPEL, A. W. AND GEORGE, L. 2001. Optimal spilling for CISC machines with few registers. In *PLDI'01*. Snowbird, Utah, USA, 243–253.

ARM. 1998. Document No. ARM DDI 0084D, ARM Ltd. ARM7TDMI-S data sheet.

AVISSAR, O., BARUA, R., AND STEWART, D. 2002. An optimal memory allocation scheme for scratch-pad-based embedded systems. *ACM Trans. Embed. Comput. Syst. 1,* 1, 6–26.

BERRY, M., CHEN, D., KOSS, P., KUCK, D., LO, S., PANG, Y., POINTER, L., ROLOFF, R., SAMEH, A., CLEMENTI, E., CHIN, S., SCHNEIDER, D., FOX, G., MESSINA, P., WALKER, D., HSIUNG, C., SCHWARZMEIER, J., LUE, K., ORSZAG, S., SEIDL, F., JOHNSON, O., AND GOODRUM, R. 1988. The perfect club benchmarks: Effective performance evaluation of supercomputers. *International Journal of Supercomputer Applications 3*, 5–40.

BOISSINOT, B., DARTE, A., DE DINECHIN, B. D., GUILLON, C., AND RASTELLO, F. 2009. Revisiting out-of-SSA translation for correctness, code quality and efficiency. In *CGO'09*. 114–125.

BOUCHEZ, F., DARTE, A., GUILLON, C., AND RASTELLO, F. 2006a. Register allocation: What does the NP-completeness proof of Chaitin et al. really prove? In *WDDD'06* (July). Boston, MA.

BOUCHEZ, F., DARTE, A., GUILLON, C., AND RASTELLO, F. 2006b. Register allocation: What does the NP-completeness proof of Chaitin et al. really prove? or revisiting register allocation: Why and how. In *LCPC'06* (Nov.). LNCS. Springer Verlag, New Orleans, Louisiana.

BOUCHEZ, F., DARTE, A., AND RASTELLO, F. 2007. On the complexity of register coalescing. In *CGO'07* (Mar.).

BOUCHEZ, F., DARTE, A., AND RASTELLO, F. 2008. Advanced conservative and optimistic register coalescing. In *CASES'08*. 147–156.

BRAUN, M. AND HACK, S. 2009. Register spilling and live-range splitting for ssa-form programs. In *Compiler Construction*. Lecture Notes in Computer Science Series, vol. 5501. Springer Berlin / Heidelberg, 174–189.

BRISK, P., DABIRI, F., JAFARI, R., AND SARRAFZADEH, M. 2006. Optimal register sharing for high-level synthesis of ssa form programs. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on 25,* 5, 772 – 779.

BURNS, M., PRIER, G., MIRKOVIC, J., AND REIHER, P. 2003. Implementing address assurance in the Intel IXP.

CHEN, L. 1992. Optimal parallel time bounds for the maximum clique problem on intervals. *Inf. Process. Lett. 42,* 4, 197–201.

DIOUF, B., OZTURK, O., AND COHEN, A. 2009. Optimizing local memory allocation and assignment through a decoupled approach. In *The 22nd International Workshop on Languages and Compilers for Parallel Computing (LCPC 2009)*. Newark, Delaware, USA.

DOMINGUEZ, A., NGUYEN, N., AND BARUA, R. K. 2007. Recursive function data allocation to scratch-pad memory. In *CASES'07*. 65–74.

FABRI, J. 1979. Automatic storage optimization. In *ACM Symp. on Compiler Construction*. 83–91.

GOLUMBIC, M. C. 2004. *Algorithmic Graph Theory and Perfect Graphs (Annals of Discrete Mathematics, Vol 57)*. North-Holland Publishing Co., Amsterdam, The Netherlands, The Netherlands.

HACK, S., GRUND, D., AND GOOS, G. 2005. Towards register allocation for programs in ssa-form. Tech. Rep. 2005-27, Universität Karlsruhe. September.

HACK, S., GRUND, D., AND GOOS, G. 2006. Register allocation for programs in SSA-form. In *CC'06*. 247–262.

INSTRUMENTS, T. 1997. TMS370Cx7x 8-bit microcontroller, Texas Instruments.

ISSENIN, I., BROCKMEYER, E., MIRANDA, M., AND DUTT, N. 2007. DRDU: A data reuse analysis technique for efficient scratch-pad memory management. *ACM Trans. Des. Autom. Electron. Syst. 12,* 2, 15.

KAHLE, J. A., DAY, M. N., HOFSTEE, H. P., JOHNS, C. R., MAEURER, T. R., AND SHIPPY, D. 2005. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development 49,* 4/5.

KANDEMIR, M., RAMANUJAM, J., IRWIN, J., VIJAYKRISHNAN, N., KADAYIF, I., AND PARIKH, A. 2001. Dynamic management of scratch-pad memory space. In *DAC'01*. 690–695.

LEE, C. G. 1998. UTDSP benchmarks.

LEE, J. K., PALSBERG, J., AND PEREIRA, F. M. Q. 2008. Aliased register allocation for straight-line programs is NP-compl ete. *Theoretical Computer Science 407*, 258–273. Preliminary version in Proceedings of ICALP'07, 34th International Colloquium on Automata, Languages and Programming, pages 680–691, Wroclaw, Poland, July 2007.

LI, L., FENG, H., AND XUE, J. 2009. Compiler-directed scratchpad memory management via graph coloring. *ACM Trans. Archit. Code Optim. 6,* 9:1–9:17.

LI, L., GAO, L., AND XUE, J. 2005. Memory coloring: A compiler approach for scratchpad memory management. In *PACT'05*. 329–338.

LI, L., XUE, J., AND KNOOP, J. 2011. Scratchpad memory allocation for data aggregates via interval coloring in superperfect graphs. *ACM Trans. Embed. Comput. Syst. 10*, 28:1–28:42.

MOTOROLA. 1998. M-CORE – MMC2001 reference manual, Motorola Corporation.

NVIDIA. 2008. NVIDIA unified architecture GeForce 8800 GT.

PEREIRA, F. AND PALSBERG, J. 2009. Ssa elimination after register allocation. In *Compiler Construction*, O. Moor and M. Schwartzbach, Eds. Lecture Notes in Computer Science Series, vol. 5501. Springer Berlin Heidelberg, 158–173.

PEREIRA, F. M. Q. AND PALSBERG, J. 2005. Register allocation via coloring of chordal graphs. In *In Proceedings of APLAS05, Asian Symposium on Programming Languages and Systems*. 315–329.

POUCHET, L. N. 2012. Polybench/c, the polyhedral benchmark suite. `http://www.cse.ohio-state.edu/~pouchet/software/polybench/`.

QUINTÃO PEREIRA, F. M. AND PALSBERG, J. 2008. Register allocation by puzzle solving. *SIGPLAN Not. 43,* 6, 216–226.

SAHA, A., PAL, M., AND PAL, T. K. 2007. Selection of programme slots of television channels for giving advertisement: A graph theoretic approach. *Inf. Sci. 177,* 12, 2480–2492.

SARKAR, V. AND BARIK, R. 2007. Extended linear scan: An alternate foundation for global register allocation. In *CC'07*, S. Krishnamurthi and M. Odersky, Eds. Lecture Notes in Computer Science Series, vol. 4420. Springer, 141–155.

SJÖDIN, J. AND VON PLATEN, C. 2001. Storage allocation for embedded processors. In *Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems*. CASES '01. ACM, New York, NY, USA, 15–23.

STEINKE, S., WEHMEYER, L., LEE, B., AND MARWEDEL, P. 2002. Assigning program and data objects to scratchpad for energy reduction. In *Proceedings of the conference on Design, automation and test in Europe*. DATE '02. IEEE Computer Society, Washington, DC, USA, 409–.

UDAYAKUMARAN, S. AND BARUA, R. 2003. Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. In *CASES'03*. 276–286.

UDAYAKUMARAN, S., DOMINGUEZ, A., AND BARUA, R. 2006. Dynamic allocation for scratch-pad memory using compile-time decisions. *ACM Trans. Embed. Comput. Syst. 5,* 2, 472–511.