Optimizing the Efficiency of Parameterized Local Search within Global Search: A Preliminary Study

Eckart Zitzler
Computer Engineering Laboratory
Swiss Federal Institute of Technology
Zurich, Switzerland
zitzler@tik.ee.ethz.ch

Jürgen Teich
Computer Engineering Laboratory
University of Paderborn
Paderborn, Germany
teich@date.uni-paderborn.de

Shuvra S. Bhattacharyya
ECE Department and UMIACS
University of Maryland
College Park, USA
ssb@eng.umd.edu

Abstract- Application-specific, parameterized local search algorithms (PLSAs), in which optimization accuracy can be traded-off with run-time, arise naturally in many optimization contexts. We introduce a novel approach, called simulated heating, for systematically integrating parameterized local search into global search algorithms (GSAs) in general and evolutionary algorithms in particular. Using the framework of simulated heating, we investigate both static and dynamic strategies for systematically managing the trade-off between PLSA accuracy and optimization effort. We show quantitatively that careful management of this trade-off is necessary to achieve the full potential of a GSA/PLSA combination. Furthermore, we provide preliminary results which demonstrate the effectiveness of our simulated heating techniques in the context of code optimization for embedded software implementation, a practical problem that involves vast and complex search spaces.

1 Motivation

For many useful optimization problems, including a significant proportion of those arising in electronic design automation, efficient algorithms exist for refining arbitrary points in the search space into better solutions. Such algorithms are called *local search algorithms* because they define neighborhoods, typically based on initial "coarse" solutions, in which to search for optima. Many of these algorithms are parameterizable in nature. Based on the values of one or more algorithm parameters, such a *parameterized local search algorithm* (*PLSA*) can trade-off time/space complexity for optimization accuracy (quality of the optimized result).

Local search techniques can often be incorporated naturally into global search algorithms (GSAs) in order to increase the effectiveness of optimization. This has the potential to exploit the complementary advantages of GSAs like evolutionary algorithms (generality, robustness, global search efficiency), and problem-specific PLSAs (rapid convergence toward local minima). For instance, in the field of evolutionary computation hybridization seems to be common for real-world applications [GV99] and a lot of evolutionary algorithm/local search method combinations can be found in the literature, e.g., [Dav91, IM96, RDSW99, MF99, ZTB99].

When employing PLSAs, however, the question is how to use the computational resources most efficiently, i.e., either one is interested in finding a solution of sufficient quality in minimum time or the goal is to generate a maximum quality solution within a specified time [GV99]. Here, we consider the later case. Given a fixed optimization time budget, which is a realistic assumption in practical optimization applications, what proportion of time should be allocated for local search with the PLSA at each optimization step? More time alloted to each PLSA invocation implies more thorough local optimization at the expense of a smaller number of achievable function evaluations (e.g., smaller numbers of generations explored with evolutionary methods), and vice-versa. Arbitrary management of this trade-off between accuracy and run-time of the PLSA is likely to produce highly suboptimal results. Furthermore, the proportion of time that should be allocated to each call of the local search procedure is likely to be highly problem-specific and even instance-specific. Thus, dynamic adaptive approaches may be more desirable than static approaches.

In this paper, we introduce a novel technique, called *simulated heating*, that systematically incorporates parameterized local search into the framework of global search. The idea is to increase the time alloted to each PLSA invocation during the optimization process (low accuracy of the PLSA at the beginning and high accuracy at the end). In contrast to other studies which investigated the behavior of GSA/PLSA hybrids [Whi95, GV99], the focus is here on the PLSA and the corresponding accuracy/run-time trade-offs to be made rather than on the question of how to optimally share the available computational resources among GSA and PLSA. That is the time consumed by the GSA is not taken into account, the maximum time budget only limits the time resources provided for the PLSA; as soon as this budget is used up the optimization process is stopped. Within the context of simulated heating optimization, we consider both static and dynamic strategies for systematically increasing the PLSA accuracy and the corresponding optimization effort. Our goals are to show that careful management of this trade-off is necessary to achieve the full potential of a GSA/PLSA combination, and to develop an efficient strategy for achieving this trade-off management. We demonstrate our techniques in the context of code optimization for embedded software implementation, where an evolutionary algorithm is used as GSA. This is a critical optimization problem in electronic design automation, and one that involves vast and complex search spaces.

The organization of the paper is as follows. First, we develop the motivation for careful management of the accuracy/run-time trade-offs that underlie PLSA techniques. We then develop a general framework for combining global search techniques with PLSAs in a systematic fashion. Subsequently, we introduce the principles of simulated heating, and develop first static, and then dynamic heating schemes to apply within the broad, flexible framework offered by our simulated heating concept. Subsequently, we examine a highly-relevant and complex optimization problem, called the MCMP, from the domain of embedded software optimization; we define how a PLSA emerges naturally from within this optimization context; and based on this PLSA, we develop a specific simulated heating formulation to attack the MCMP. Finally, we discuss our experimental methodology, present an enlightening set of experimental results, and summarize with our conclusions and observations on future research directions.

2 Optimization Scenario

Suppose that we have a GSA G operating on a set of solution candidates and a PLSA $L(p)$, where p is the parameter of the local search procedure.[1] Let

- $C(p)$ denote the complexity (worst-case run-time) of L for the parameter choice p,
- $A(p)$ be the accuracy (effectiveness) of L with regard to p, and
- R denote the set of permissible values for parameter p. Typically, R may be described by an interval $[p_{\min}, \ldots, p_{\max}] \cap \mathbf{R}$ where $\mathbf{R}$ denotes the set of reals.

Generally, it is very difficult if not impossible to analytically determine the functions $C(p)$ and $A(p)$, but these functions are useful conceptual tools in discussing the problem of designing cooperating GSA/PLSA combinations. Our proposed techniques do not require these functions to be known. The only assumption we make here is that $C(p)$ and $A(p)$ are monotonically non-decreasing regarding R. As a consequence, there is a trade-off: when $A(p)$ is low, refinement is generally not too much, but not much time is consumed ($C(p)$ is also low). Conversely higher $A(p)$ requires higher computational cost $C(p)$.

The following G/L hybrid is taken as the basis for the optimization scenario considered in this paper:

Input:	N	(size of the solution candidate set)
	$T_{\max}$	(maximum time budget)
Output:	s	(best solution found)

[1] For simplicity it is assumed here that p is a scalar rather than a vector of parameters.

Step 1: **Initialization**: Create an initial multi-set S containing N randomly generated solution candidates. Set $T = 0$ (time used) and $t = 0$ (iterations performed).

Step 2: **Parameter adaptation**: Choose $p \in R$ according to a given scheme H: $p = H(t)$.

Step 3: **Local search**: Apply L with parameter p to each $s \in S$ and assign it a quality (fitness) $F(s)$. Set $T = T + N \cdot C(p)$.

Step 4: **Termination**: If $T > T_{\max}$ then go to Step 6.

Step 5: **Global search**: Based on S and F, generate a new set S' of solution candidates using G. Set $S = S'$ and increase the iteration counter t. Go to Step 2.

Step 6: **Output**: Apply L with parameter $p_{\max}$ to the best solution in S regarding F; the resulting solution s is the outcome of the algorithm.

The GSA G operates on a set of N solution candidates (N may be equal to one for, e.g., simulated annealing or greater than one for, e.g., an evolutionary algorithm); per optimization step, it creates a new set of solution candidates depending on the previous solution set and the quality function associated with it. The PLSA L is used to refine and/or to evaluate the solution candidates generated by G; its parameter p is adapted in each iteration according to a predefined scheme. Furthermore, a fixed time limit determines how many iterations of the main loop of the hybrid may be performed. As we are mainly interested here in trading-off accuracy and runtime of L, only the optimization time needed by L is taken into account when calculating the remaining time budget. At the end, when the given time limit is exceeded, L is applied to the best solution in S using maximum accuracy $A(p_{\max})$.

In the following, we discuss different adaptation schemes H how to systematically vary p during the optimization run in order to use the given time resources most efficiently. The underlying approach is called simulated heating.

3 Simulated Heating

3.1 Basic Principles

The idea of simulated heating can be summarized as follows: Instead of keeping p constant for the entire optimization process, we start with a low value (leading to low $C(p)$ and $A(p)$) and increase it at certain points in time (which in turn increases $C(p)$ and $A(p)$). What is the benefit of doing so? If p is fixed, each iteration of the overall optimization procedure requires approximately the same amount of time. This is depicted in Figure 1 by the dashed line. In contrast, with simulated heating the time resources used per iteration are lower at the beginning and higher at the end (cf. the dotted line in Figure 1). That is, in the first half of the time a greater number of iterations is performed than in the second half with regard to a single run.

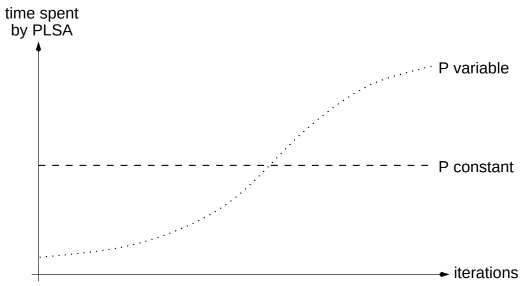

Figure 1: Comparison of two basic principles: fixed p versus variable p. When using a fixed p, the time consumed by the PLSA per iteration is constant. In contrast, variations of p may lead to a non-constant function regarding the time resources needed per distinct iteration.

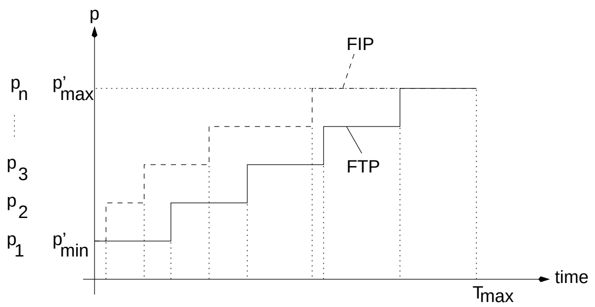

Figure 2: Illustration of two different heating schemes investigated in the context of this paper: FIP (fixed number of iterations per parameter) and FTP (fixed time per parameter).

In other words, the goal is to focus on the global search at the beginning and to find promising regions first; for this phase, L runs with low accuracy, which in turn allows an greater number of optimization steps of G. Afterwards, more time is spent by L in order to improve the solutions found so far and/or to assess them more accurately. As a consequence, fewer global search operations are possible during this phase of optimization. Since $A(p)$ is systematically increased in the course of time, we use the term simulated heating for this approach by analogy to simulated annealing where a 'temperature' is continuously decreased according to a given cooling scheme.

Definition 1 [Heating scheme] *Let* $H : \mathbf{N}_0 \mapsto R$ *with* R *defined by the set of reals in the interval* $[p_{\min}, \dots, p_{\max}]$ *be a function which specifies the PLSA parameter* $p = H(t)$ *to be used during local search at iteration* t *of the general GSA/PLSA hybrid.* H *is called* heating scheme*, if for all* $t_1, t_2 \in \mathbf{N}_0$ *with* $t_1 \leq t_2$ *and* $p_1 = H(t_1)$*,* $p_2 = H(t_2)$*:* $p_{\min} \leq p_1 \leq p_2 \leq p_{\max}$.

Figure 2 shows two different heating schemes. Both schemes have in common that the number and values of parameters considered over time between $T = 0$ and $T = T_{\max}$ is the same, and that the value of p is increased by a constant each time the PLSA parameter is adapted.

In realizing simulated heating, there are two fundamental ways of implementing a heating scheme H: by static or dynamic adaptation mechanisms. In the first case, it is assumed that the heating scheme is fixed per optimization run. Thus, it may be computed at compile-time or directly before the actual optimization (Step 1 of the general hybrid). In the latter case, the heating scheme evolves during run-time, i.e., is computed during the optimization run. Hence, it may vary for different runs.

Another orthogonal classification that will be investigated in the following is whether i) an equal number of iterations is performed for each parameter investigated in the parameter interval $R' \subseteq R$ or ii) constant optimization time is spent for each parameter considered. We call the first class of schemes FIP (*fixed number of iterations per parameter*) and the second class FTP (*fixed time per parameter*). With FTP, the optimization time is spread equally for each parameter. As a consequence, the number of iterations that may be performed for each fixed parameter decreases for higher values of p, see also Figure 2.

All four algorithmic variants of simulated heating (static vs. dynamic and FIP vs. FTP) will be compared in Section 5 using an application from the domain of embedded software optimization. However, in general there are too many different heating schemes that may be analyzed. For the following experiments, we restricted ourselves to heating schemes that, given a parameter interval $R' \subseteq R$, and a number n of different parameters, select the parameter uniformly from over R' as follows: Let R' be the set of reals in the interval $[p'_{\min}, \cdots, p'_{\max}]$. Then, $p_1 = p'_{\min}$, $p_2 = p'_{\min} + \epsilon$, ..., $p_n = p_1 + (n-1)\epsilon = p'_{\max}$. Hence, the step size ϵ for increasing the parameter may be statically approximated as[2]

$$\epsilon = (p'_{\max} - p'_{\min})/(n-1)$$

3.2 Static Heating Schemes

First, we consider static heating schemes. With the uniform distribution of parameters as presented above, it remains to determine the points in time where to switch from one parameter to the next (see Step 2 of our optimization scenario).

We will see that it is necessary to estimate the computational cost $C(p)$ of each parameter p to be considered. This can be done by applying the PLSA to randomly chosen solutions and estimating from the resulting run-times the time $C(p)$ needed for one PLSA invocation with parameter p.

3.2.1 Fixed Number of Iterations Per Parameter (FIP)

In this strategy (FIP), the parameter p_i is constant for exactly $t_i = t_p$ iterations. The question is therefore, how many iterations t_p may be performed per parameter within the time budget $T_{\max}$.

Having the constraint

$$T_{\max} \geq t_p N C(p_1) + t_p N C(p_2) + \ldots + t_p N C(p_n)$$

[2] In case only integer parameters are allowed, it is appropriate to take the ceiling function ($\epsilon' = \lceil \epsilon \rceil$).

we obtain t_p with

$$t_p = \left\lfloor \frac{T_{\max}}{N \sum_{i=1}^{n} C(p_i)} \right\rfloor$$

as the number of iterations assigned to each p_i.

3.2.2 Fixed Amount of Time Per Parameter (FTP)

For the FTP strategy, the points in time where to increase p are equi-distant and may be simply computed as follows. Obviously, the time-budget, when equally split between n parameters becomes $T_p = T_{\max}/n$ per parameter. Hence, the number of iterations t_i that may be performed using parameter $p_i, i = 1, \ldots, n$ is restricted by

$$t_i N C(p_i) \leq T_p \quad \forall i = 1, \ldots, n$$

Thus, we obtain

$$t_i = \left\lfloor \frac{T_{\max}}{n N C(p_i)} \right\rfloor$$

as the maximum number of iterations that may be computed using parameter p_i in order to stay within the given time budget.

3.3 Dynamic Heating Schemes

Here, we assume that the choice of the parameters is the same as for the static schemes: Given an interval $R' \subseteq R$, n parameters are uniformly chosen over R'.

3.3.1 Fixed Number of Iterations Per Parameter (FIP)

Here, the next parameter is taken when for a number t_{stag} of iterations the quality of the best solution in the solution candidate set has not improved. As a consequence, for each parameter a different amount of time may be considered until the stagnation condition is fulfilled.

3.3.2 Fixed Amount of Time Per Parameter (FTP)

In this case, the next parameter is taken when for T_{stag} seconds the quality of the best solution in the solution candidate set has not improved. As a consequence, for each parameter a different number of iterations may be considered until the stagnation condition is fulfilled.

4 Application: Embedded Software Optimization

4.1 Background

To help manage rapidly-escalating complexity in embedded digital signal processing (DSP) applications, DSP system designers have increasingly been employing high-level, graphical design environments in which system specification is based on hierarchical dataflow graphs [BML96]. Dataflow-based computational models provide block-diagram semantics that are both intuitive to DSP system designers, and efficient from the point of view of embedded software synthesis. In dataflow, a computational specification is represented as a directed graph in which vertices (*actors*) specify computational functions of arbitrary complexity, and edges specify communication between functions.

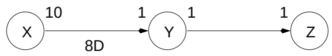

Figure 3: A simple SDF graph.

A dataflow edge represents a FIFO (first-in-first-out) queue that buffers data as it passes from the output of one actor to the input of another. When dataflow graphs are used to represent signal processing applications, a dataflow edge e has a non-negative integer *delay* $\delta(e)$ associated with it. The delay of an edge gives the number of initial data values that are queued on the edge.

Synchronous dataflow (SDF) [LM87] is the simplest and most popular form of dataflow for DSP. The SDF model imposes the restriction that the number of data values produced by an actor onto each output edge is constant, and similarly, the number of data values consumed by an actor from each input edge is constant.

A simple example of an SDF graph is shown in Figure 3. Each edge is annotated with the number of tokens produced by the source actor and the number consumed by the sink actor. The “8D” below edge (X, Y) denotes a delay of magnitude 8.

A *periodic schedule* for an SDF graph is a schedule that invokes each actor at least once, does not deadlock, and produces no net change in the number of tokens queued on each edge. An SDF graph is compiled by encapsulating the code for a period schedule within an infinite loop. Given an SDF graph, the minimum number of times $q(A)$ required to invoke each actor A in a periodic schedule can be determined using an efficient algorithm [LM87]. For example, for Figure 3, we have $q(X) = 1, q(Y) = 10, q(Z) = 10$.

A software synthesis tool generates application programs by piecing together code modules from a predefined library of software building blocks. These code modules are defined as specifications in the target language of the synthesis tool. To avoid subroutine overhead, many SDF-based design systems use a model of synthesis called *inline threading*. Given an SDF representation of a block-diagram program specification, an inline threading tool begins by constructing a periodic schedule. The synthesis tool then steps through the schedule and for each actor instance A that it encounters, it inserts the associated code module M_A from the given library. The sequence of code modules and subroutine calls that is generated from a dataflow graph is processed by a buffer management phase that inserts the necessary target program statements to route data appropriately between actors.

The scheduling phase has a large impact on the memory requirement of the final implementation. The key components of this memory requirement are the code size cost (the sum of the code sizes of all inlined modules, and of all inter-actor looping constructs), and the buffering cost (the amount of memory allocated to accommodate inter-actor data transfers). Even for a simple SDF graph, the underlying range of trade-offs may be very complex. For example, consider the SDF graph in Figure 3. Two possible periodic schedules for this graph are $S_1 = (8YZ)X(2YZ)$, and $S_2 = X(10YZ)$. Here, a parenthesized term (*schedule loop*) of the form $(nT_1T2 \dots T_m)$ specifies the successive repetition n times of the subschedule $T_1T_2 \dots T_m$. A schedule that contains zero or more schedule loops is called a *looped schedule*, and a schedule that contains exactly zero schedule loops is called a *flat schedule* (thus, a flat schedule is a looped schedule, but not vice-versa). Two looped schedules are *sequence-equivalent* if they represent the same sequence of actor invocations. For example, the looped schedules $(2(2AB))$, $(4AB)$, and $ABABABAB$ are all sequence-equivalent.

The *code size costs* for schedules S_1 and S_2 can be expressed, respectively, as $\kappa(X) + 2\kappa(Y) + 2\kappa(Z) + 2L_c$, and $\kappa(X) + \kappa(Y) + \kappa(Z) + L_c$, where L_c denotes the processor-dependent, code size overhead of a software looping construct, and $\kappa(A)$ denotes the program memory cost of the library code module for an actor A. The code size of schedule S_1 is larger because it contains more "actor appearances" than schedule S_2 (e.g., actor Y appears twice in S_1 vs. only once in S_2), and S_1 also contains more schedule loops (2 vs. 1). The *buffering cost* of a schedule is computed as the sum over all edges e of the maximum number of buffered (produced, but not yet consumed) tokens that coexist on e throughout execution of the schedule. Thus, the buffering costs of S_1 and S_2 are 11 and 19, respectively. The *memory cost* of a schedule is the sum of its code size and buffering costs. Thus, depending on the relative magnitudes of $\kappa(X)$, $\kappa(Y)$, $\kappa(Z)$ and L_c, either S_1 or S_2 may have lower memory cost.

4.2 Memory Cost Minimization Problem (MCMP)

The *memory cost minimization problem* (*MCMP*) is the problem of computing a looped schedule that minimizes the memory cost for a given SDF graph, and a given set of actor and loop code sizes.

It has been shown that the problem of constructing a schedule that minimizes the buffering cost over all periodic schedules is NP-complete [BML96]. This result is easily adapted to establish intractability of the memory cost minimization problem. A related problem is the *schedule looping problem*, which is the problem of deriving a minimum code size, sequence-equivalent looped schedule from a given flat schedule. This problem, unlike the MCMP, is tractable, and can be solved using an $O(n^4)$ algorithm called *CDPPO* (code size dynamic programming post optimization), where n is the schedule length [BML95]. As we will explain in Section 4.3.2, the CDPPO algorithm can be formulated naturally as a PLSA with a single parameter such that accuracy and run-time both increase *monotonically* with the parameter value.

Note that since the output schedule is sequence-equivalent to the input schedule, CDPPO does not change the buffering cost—only the code size cost may change—and in general, the code size cost is reduced or stays the same (usually a significant reduction is observed [BML95]). CDPPO can thus be employed as a local search algorithm within any global search algorithm that operates at the level of flat schedules. In the context of multi-objective (Pareto front computation) data memory, code size, and execution time optimization, we have, in this manner, combined CDPPO with an evolutionary algorithm [ZTB99, ZTB00]. However, in this previous work, we uniformly applied the "full-strength" (maximum accuracy/maximum run-time) form of CDPPO, and as conventionally done with local search techniques, did not explore application of its PLSA form. Our objective with CDPPO in this work is very different: we seek to understand the effectiveness of its PLSA formulation, and the manner in which the associated *monotonic accuracy/run-time* trade-off should be managed during optimization. Thus, since multi-objective optimization scenarios involve more complex (non-monotonic) relationships between the CDPPO PLSA parameter and the associated optimization objective, we focus on a one-dimensional objective function (memory cost) in this work. A useful direction for further study is the exploration of PLSAs in the context of multi-objective optimization.

4.3 Implementation

To tackle the MCMP we use a GSA/PLSA hybrid as discussed in Section 2 where an evolutionary algorithm is the GSA and CDPPO is the PLSA. Our implementation is a "lean" version of the one presented in [ZTB99, ZTB00]. Here, a simpler encoding (only the schedule is represented) and another fitness function (single instead of multiple objectives) are considered.

In the following, the evolutionary algorithm and the parameterized CDPPO are sketched. Details can be found in [ZTB00].

4.3.1 GSA: Evolutionary Algorithm

Each solution s is encoded by an integer vector which represents the corresponding schedule, i.e., the order of actor firings. Its length is fixed because the number of firings for each actor is known a priori (repetition vector). The decoding process which takes place in the local search/evaluation phase (Step 3 of the general algorithm on page 2) is as follows:

- first a repair procedure is invoked which transforms the encoded actor firing sequence into a valid flat schedule,
- then the parameterized CDPPO is applied to the resulting flat schedule in order to compute a (sub)optimal looping, and

- afterwards the data requirement (buffering cost) $D(s)$ and the program requirement (code size cost) $P(s)$ of the software implementation represented by the looped schedule are calculated based on a certain processor model.

Finally, both $D(s)$ and $P(s)$ are normalized (the minimum and maximum values $D_{\min}$ and $D_{\max}$ respectively $P_{\min}$ and $P_{\max}$ for the distinct objectives can be determined beforehand) and a fitness is assigned to the solution s according to the formula below:

$$F(s) = 0.5\frac{D(s) - D_{\min}}{D_{\max} - D_{\min}} + 0.5\frac{P(s) - P_{\min}}{P_{\max} - P_{\min}}$$

Note that the fitness values are to be minimized here.

Concerning the genetic operators, binary tournament selection, order-based uniform crossover and scramble sublist mutation [Dav91] are used. Moreover, a simple elitism mechanism is incorporated which ensures that the best solution in the population always survives. The interested reader is referred to [ZTB99] for further information.

4.3.2 PLSA: Parameterized CDPPO

The ("unparameterized") CDPPO algorithm was first proposed in [BML95]. CDPPO computes an optimal parenthesization in a bottom-up fashion, which is analogous to dynamic programming techniques for matrix-chain multiplication [CLR92]. Given, an SDF graph $G = (V, E)$ and an actor invocation sequence (flat schedule) $f_1, f_2, \ldots, f_n$, where each $f_i \in V$, CDPPO first examines all 2-invocation *sub-chains* $(f_1, f_2), (f_2, f_3), \ldots, (f_{n-1}, f_n)$ to determine an optimally-compact looping structure (*subschedule*) for each of these sub-chains. For a 2-invocation sub-chain (f_i, f_{i+1}), the most compact subschedule is easily determined: if $f_i = f_{i+1}$, then $(2f_i)$ is the most compact subschedule, otherwise the original (unmodified) subschedule $f_i f_{i+1}$ is the most compact. After the optimal 2-node subschedules are computed in this manner, these subschedules are used to determine optimal 3-node subschedules (optimal looping structures for subschedules of the form f_i, f_{i+1}, f_{i+2}); the 2- and 3-node subschedules are then used to determine optimal 4-node subschedules, and so on until the n-node optimal subschedule is computed, which gives a minimum code size implementation of the input invocation sequence $f_1, f_2, \ldots, f_n$. Further details on CDPPO can be found in [ZTB00].

Due to its high complexity, CDPPO can require significant computational resources for a single application — e.g., we have commonly observed run-times on the order of 30-40 seconds for practical applications. In the context of global search techniques, such performance can greatly limit the number of neighborhoods (flat schedules) in the search space that are sampled. To address this limitation, however, a simple and effective parameterization emerges: we simply set a threshold M on the maximum sub-chain (subschedule) size that optimization is attempted on. This threshold becomes the parameter of the resulting *parameterized CDPPO (PCDPPO)* algorithm. Thus, PCDPPO operates as CDPPO does, except that the subschedule optimization process stops after all optimal M-node subschedules have been completed. Consequently, looping structures are exploited only within actor invocation subsequences having length less than or equal to M. Repetitive firing sequences that span more than M actor invocations remain "unlooped."

In summary PCDPPO is a parameterized adaptation of CDPPO for addressing the schedule looping problem. The run-time and accuracy of PCDPPO are both monotonically nondecreasing functions of the algorithm "threshold" parameter M. In the context of the memory minimization problem, PCDPPO is a genuine PLSA.

5 Experiments

5.1 Methodology

In order to investigate the efficiency of the proposed simulated heating approach, we evaluated its performance on a complex, real-life application benchmark, which we refer to as the *CD2DAT* benchmark. This benchmark performs a sample-rate conversion from a compact disk (CD) player output (44.1 kHz) to to a digital audio tape (DAT) input (48 kHz). Due to the extensive multirate processing involved, this application requires over 600 actor invocations per flat schedule, and consequently, the quartic complexity of the original (full-strength) CDPPO becomes heavily cumbersome for probabilistic global search techniques. The CD2DAT application has been used extensively as a benchmark to evaluate SDF-based software optimization techniques (e.g., see [BML96, BML99]). For details on sample-rate conversion, and for further details on the complex, SDF structure of the CD2DAT benchmark, we refer the reader to [Vai93], and [BML99], respectively.

We considered five types of parameter adaptation for the CD2DAT application:

PC: p constant over the entire optimization run,

SI: static heating with fixed number of iterations (FIP),

ST: static heating with fixed amount of time (FTP),

DI: dynamic heating with fixed number of iterations (FIP),

DT: dynamic heating with fixed amount of time (FTP).

For the first type, five different values for p (uniformly distributed over R) were tested: $1, 153, 305, 457, 612$.[3] In the case of static and dynamic heating, we varied the parameter interval R': one time R' was set to $[1, 612]$, another time a smaller interval $R' = [1, 153]$ was taken; however, independent of R', the number of parameters considered was $n = 5$. Altogether, this resulted in 13 different adaptation schemes.

[3] For the CD2DAT example, R is $[1, 612]$.

heating scheme		results				
type	p / R'	min		median		max
PC	1	0.3120	0.3289	**0.3394**	0.3465	0.3549
PC	153	0.3061	0.3273	**0.3308**	0.3389	0.3512
PC	305	0.3599	0.3605	**0.3637**	0.3651	0.3693
PC	457	0.3574	0.3611	**0.3622**	0.3670	0.3692
PC	612	0.3482	0.3663	**0.3692**	0.3693	0.3746
SI	$[1, 612]$	0.3529	0.3548	**0.3558**	0.3566	0.3621
SI	$[1, 153]$	0.2757	0.2831	**0.2848**	0.2903	0.3045
ST	$[1, 612]$	0.2871	0.2909	**0.3024**	0.3071	0.3106
ST	$[1, 153]$	0.2405	0.2434	**0.2609**	0.2680	0.2708
DI	$[1, 612]$	0.2810	0.2839	**0.2992**	0.3041	0.3082
DI	$[1, 153]$	0.2723	0.2737	**0.2739**	0.2793	0.2962
DT	$[1, 612]$	0.2914	0.2974	**0.2985**	0.3022	0.3081
DT	$[1, 153]$	0.2031	0.2554	**0.2558**	0.2660	0.2725

Table 1: Fitness values of the best solutions found by the 13 different heating schemes in five optimization runs.

For each of the schemes, five optimization runs were performed on a Sun ULTRA 60 using a time budget $T_{\max}$ of 5 hours. To avoid random effects, all schemes were tested on the same set of initial populations. Concerning the parameters of the evolutionary algorithm, the values from [ZTB99] were taken:

population size N	:	100
crossover rate p_c	:	0.8
mutation rate p_m	:	0.1

Finally, t_{stag} was set to 10 iterations for DI and T_{stag} to 900 seconds for DT. Both parameters were chosen rather based on intuition than on experimental results.

5.2 Comparison of Different Heating Schemes

The results which make up Table 1 indicate that the choice of the parameter p does affect the outcome of the optimization process.

When keeping p constant, obviously smaller p values (1, 153) lead to better solutions than greater values (305, 457, 612). This is due to the larger number of iterations that can be performed for low complexities $C(p)$ (cf. Table 2). It seems that there is a point from which increasing p decreases the performance of the hybrid algorithm. As illustrated in Figure 4, continuously increasing p starting from $p = p_{\min}$ also increases the accuracy $A(p)$ of the PLSA and therefore the effectiveness of the overall algorithm. However, when a certain run-time complexity $C(p_{\mathrm{opt}})$ of the PLSA is reached, the benefit of higher accuracy may be outweighed by the disadvantage that the number of iterations which can be explored is smaller. As a consequence, values greater than p_{opt} may reduce the overall performance as the number of iterations is too low. Nevertheless, to review this assumption several p values must be compared for a larger number of runs and for different applications. This is the subject of further work.

Comparing the simulated heating approach for $R' = [1, 612]$ with the conventional method PC, the adaptation schemes ST, DI, and DT provide better results than each of the PC variants considered. Only with SI, for which the number of iterations per parameter value was fixed, the quality

heating scheme		iterations per parameter p				
type	p / R'	1	153	305	457	612
PC	1	900	0	0	0	0
PC	153	0	73	0	0	0
PC	305	0	0	22	0	0
PC	457	0	0	0	12	0
PC	612	0	0	0	0	10
SI	$[1, 612]$	4	4	4	4	4
ST	$[1, 612]$	176	14	4	2	2
DI	$[1, 612]$	99	33	12	0	0
DT	$[1, 612]$	276	11	4	1	4

Table 2: Iterations performed per parameter value for nine heating schemes. Concerning DI and DT, the numbers are related to a single optimization run; for the other four runs they look slightly different.

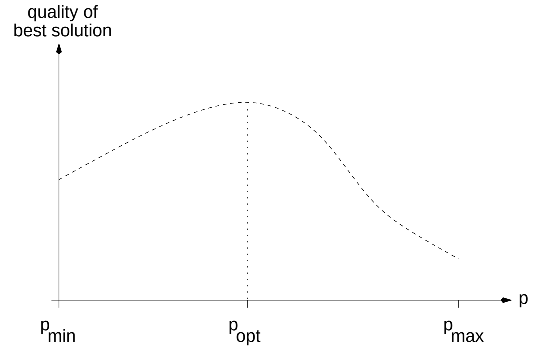

Figure 4: Relationship between the value for p and the outcome of the optimization process.

of the outcomes is slightly worse than the one achieved by PC with $p = 1$ and $p = 153$; the performance is similar to PC with $p = 305$. Taking a look at Table 2, the low number of iterations performed for the parameter values $p = 1$ and $p = 153$ may be an explanation for this phenomenon. In the case of ST, DI, and DT much more computational resources are alloted to these two parameter values.

The observation that the parameter range $R' = [1, 153]$ appears to be more promising than the entire range of permissible p values leads to the question whether the heating schemes can do better when using this reduced range. As can be seen in Table 1, this is the case. The outcomes could be improved significantly and all heating schemes outperform the different PC schemes under consideration. On average, the fitness of the best solution found by ST and DT is smaller by more than 25% in comparison to PC.

Among the heating schemes, the FTP variants ST and DT using a fixed amount of time per parameter value seem to have advantages over the FIP versions SI and DI. However, a thorough investigation is necessary to substantiate this assumption. The same holds for the comparison of static and dynamic heating schedules. The latter approach appears to achieve slightly better performance, although these preliminary experiments do not allow to arrive at a final conclusion. Nevertheless, taking the fact into account that the setting of t_{stag} and T_{stag} was not studied systematically here, there may be room for improvement with regard to the dynamic heating

heating scheme		population size		
type	p / R'	50	100	200
PC	153	0.2928	0.3061	0.3371
SI	$[1, 612]$	0.3265	0.3558	0.3676
ST	$[1, 612]$	0.3037	0.3071	0.2867
DI	$[1, 612]$	0.2788	0.2839	0.2759
DT	$[1, 612]$	0.3199	0.3022	0.2752

Table 3: Outcomes achieved using three different population sizes. Per optimization run, the corresponding number gives the fitness value of the best solution found.

schemes.

5.3 Influence of the Population Size

For the experiments presented in Section 5.2 a population size of 100 was used. In order to study the effects of the population size on the evolution process, also runs with smaller ($N = 50$) and larger ($N = 200$) populations were performed for five of the nine parameter adaptation schemes.

The outcomes of the 15 optimization runs are shown in Table 3. For PC ($p = 153$) and SI ($R' = [1, 612]$) smaller population sizes seem to be preferable; the larger number of iterations which can be explored for $N = 50$ may be an explanation for the better performance. In contrast, the heating schemes ST and DT ($R' = [1, 612]$) achieve better results when a larger population ($N = 200$) is used. In the case of DT one can observe that more than three times more computational effort is devoted to the parameter value $p = 1$ with $N = 200$ than with $N = 100$; it appears that population diversity is an important issue in this context. The situation for DI ($R' = [1, 612]$) is ambiguous.

Although these results must be interpreted with care as only one run was considered per population size/heating scheme combination, they demonstrate that there is a close relationship between the population size, the number of iterations and the value for p with regard to the quality of the outcome. As a consequence, a promising direction for future work might be to not only vary the number of iterations per parameter value but also to choose the population size depending on p (and possibly on other parameters) leading to dynamic population sizes.

6 Conclusions

This paper has motivated the importance of carefully managing the run-time/accuracy trade-offs associated with GSA/PLSA (global search/parameterized local search) hybrid algorithms, and has introduced a novel framework of simulated heating for this purpose. We have developed both static and dynamic trade-off management strategies for our simulated heating framework, and have evaluated these techniques on a complex, practical optimization problem related to embedded software implementation. Preliminary, proof-of-principle results, indicate that our approach is promising. Although our experiments have considered only a limited number of runs and only a single application (CD2DAT) was considered, we are encouraged from the significant and consistent performance improvement observed, and the high complexity (over 600 actor invocations within the schedule) of the application that we considered. Numerous useful directions for further investigation emerge from our study. These include evaluating the impact on other application systems in embedded software implementation, exploring the applicability to other important optimization problems that incorporate PLSA subsystems, and developing alternative heating schemes (e.g., exponential functions as with simulated annealing).

Acknowledgments

S. S. Bhattacharyya was supported in this work by the US National Science Foundation (CAREER, MIP9734275).

Bibliography

[BML95] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. Optimal parenthesization of lexical orderings for DSP block diagrams. In *Proceedings of the International Workshop on VLSI Signal Processing*. IEEE press, October 1995. Sakai, Osaka, Japan.

[BML96] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. *Software Synthesis from Dataflow Graphs*. Kluwer, Norwell, MA, 1996.

[BML99] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. Synthesis of embedded software from synchronous dataflow specifications. *Journal of VLSI Signal Processing Systems*, 21(2):151–166, June 1999.

[CLR92] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1992.

[Dav91] Lawrence Davis. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, New York, 1991.

[GV99] David E. Goldberg and Siegfried Voessner. Optimizing global-local search hybrids. In *GECCO'99*, volume 1, pages 220–228, San Francisco, CA, 1999. Morgan Kaufmann,.

[IM96] Hisao Ishibuchi and Tadahiko Murata. Multi-objective genetic local search algorithm. In *IEEE Conference on Evolutionary Computation (ICEC'96)*, pages 119–124, 1996.

[LM87] E. A. Lee and D. G. Messerschmitt. Synchronous dataflow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987.

[MF99] Peter Merz and Bernd Freisleben. Genetic algorithms for binary quadratic programming. In *GECCO'99*, volume 1, pages 417–424, San Francisco, CA, 1999. Morgan Kaufmann,.

[RDSW99] Mark Ryan, Justin Debuse, George Smith, and Ian Whittley. A hybrid genetic algorithm for the fixed channel assignment problem. In *GECCO'99*, volume 2, pages 1707–1714, San Francisco, CA, 1999. Morgan Kaufmann,.

[Vai93] P. P. Vaidyanathan. *Multirate Systems and Filter Banks*. Prentice Hall, 1993.

[Whi95] Darrell Whitley. Modeling hybrid genetic algorithms. In G. Winter, J. Periaux, M. Galan, and P. Cuesta, editors, *Genetic Algorithms in Engineering and Computer Science.*, pages 191–201. John Wiley, Chichester, 1995.

[ZTB99] Eckart Zitzler, Jürgen Teich, and Shuvra S. Bhattacharyya. Evolutionary algorithm based exploration of software schedules for digital signal processors. In *GECCO'99*, volume 2, pages 1762–1769, San Francisco, CA, 1999. Morgan Kaufmann.

[ZTB00] Eckart Zitzler, Jürgen Teich, and Shuvra S. Bhattacharyya. Multidimensional exploration of software implementations for DSP algorithms. *Journal of VLSI Signal Processing*, 24(1):83–98, February 2000.