درج کارای نقاط ذخیره وضعیت برنامه در سیستم‌های نهفته با قابلیت برداشت انرژی از محیط با استفاده از الگوریتم ژنتیک

سعید پناهی[1]، مرتضی محجل کفشدوز[2]، عبدالرضا رسولی کناری [3]

[1] دانشجوی کارشناسی ارشد رشته مهندسی نرم‌افزار دانشگاه صنعتی قم، قم،
panahi.s@qut.ac.ir

[2] استادیار گروه کامپیوتر، دانشکده برق و کامپیوتر دانشگاه صنعتی قم،
mohajel@qut.ac.ir

[3] استادیار گروه کامپیوتر، دانشکده برق و کامپیوتر دانشگاه صنعتی قم،
rasouli@qut.ac.ir

چکیده

برداشت انرژی[1] از محیط یکی از بهترین منابع تامین انرژی برای سیستم‌های نهفته[2] با مصرف انرژی کم است. با این وجود به دلیل ناپایدار بودن انرژی برداشت شده از محیط، در بسیاری از مواقع فرصت کافی برای اجرای یک دور کامل برنامه وجود ندارد. راه حل ارائه شده برای این منظور، استفاده از پردازنده‌های غیرفرّار[3] است. در این پردازنده‌ها با استفاده از سناریوهای مختلفی نقاط ذخیره وضعیت برنامه[4] را قرار می‌دهند تا با از دست رفتن انرژی، وضعیت برنامه حفظ شود و پس از اجرای مجدد، مبنای ادامه کار قرار گیرد و اهداف سازگاری[5] برنامه و پیشرفت رو به جلو[6] برآورده گردد.

در این مقاله ما با مدل کردن این سیستم‌ها و با استفاده از خاصیت الگوریتم ژنتیک روش جدیدی را ارائه داده‌ایم که نقاط ذخیره وضعیت برنامه را در سیستم‌های نهفته با برداشت انرژی بطور کارا درج می‌کند. آزمایش‌های ما نشان می‌دهد که روش ما در مقایسه با آخرین روش‌های موجود بطور میانگین ***تعداد نقاط ذخیره وضعیت درج شده، تعداد اجرای آن نقاط، انرژی مصرفی و انرژی تلف شده*** *را به ترتیب ۱۲، ۳۰، ۱۰ و ۴ درصد کاهش داده است و تمام* ***خطاهای ناسازگاری*** *مرتفع شده و همزمان* ***پیشرفت رو به جلوی برنامه*** *را ۱۴٫۴۱ درصد افزایش داده است.*

کلمات کلیدی

نقاط ذخیره وضعیت برنامه، سیستم‌های نهفته، پردازنده غیرفرّار، تحلیل برون‌خط[7]، برداشت انرژی، شبیه سازی، الگوریتم ژنتیک.

۱- مقدمه

- بطور معمول در سیستم‌هایی که انرژی مورد نیازشان در حد و اندازه یک یا چند میلی وات (و یا بیشتر) است، از باتری به عنوان اولین و ساده‌ترین منبع تامین انرژی استفاده می‌شود؛ در حالی که همین باتری ساده به علت وجود معایب متعدد در استفاده از آن (مانند اندازه، وزن، شارژکردن‌های مکرّر و...) به مرور جای خود را به سیستم‌های برداشت انرژی محیطی (مانند انرژی خورشیدی، انرژی باد، انرژی جنبشی و...) داده است و روز به روز استفاده از این سیستم‌ها در حال رونق و گسترش است. به دلیل ذات ناپایدار انرژی‌های برداشت شده، در دستگاه‌هایی که از این انرژی‌ها تغذیه می‌شوند به کَرّات شاهد خواهیم بود که انرژی ورودی سیستم کاهش یافته و باعث می‌شود که دستگاه با نادیده گرفتن وضعیت برنامه‌ی در حال اجرا، از ابتدا شروع به کار نماید و انرژی مصرف شده تا رسیدن به این وضعیت تلف شود که این موضوع با میزان کم انرژی برداشت شده و اتلاف آن در تناقض است؛ بنابراین کمترین میزان انرژی برداشت‌شده با ارزش است.
- به همین سبب ذخیره‌کردن وضعیت برنامه و ادامه‌ی اجرای برنامه از وضعیتی که سیستم با کمبود انرژی از آن وضعیت خارج شده است؛ از اولویت‌های پژوهشگران این حوزه قرار گرفت و در نهایت منجر به ظهور نسل جدیدی از پردازنده‌ها شد که حافظه اصلی آنها از نوع غیرفرّار است و در هنگام کمبود انرژی، امکان ذخیره‌کردن وضعیت برنامه وجود دارد.
- قرارگیری حافظه غیرفرّار در کنار حافظه فرّارِ دستگاه هایی که با انرژی‌های محیطی ناپایدار تغذیه می‌شوند، باعث بروز مشکل ناسازگاری در داده های برنامه را ایجاد می‌کند و بدلیل استفاده‌ی خاص از سیستم‌های برداشت انرژی در سامانه‌های نهفته خودکار، مانند ربات‌ها، سنسورهای بی‌سیم اینترنت اشیا و ...، وقوع کوچکترین مشکل ناسازگاری در داده‌های برنامه، سبب از کار افتادن ربات و یا ارسال داده‌های اشتباه توسط سنسورها می‌شود، که این مسئله در دنیای امروز، که در حال شکل گیری بر پایه‌ی اینترنت اشیا است، می‌تواند خطرآفرین باشد.
- در پردازنده‌های غیرفرّار، نسبت به درج نقاط ذخیره وضعیت برنامه، به عنوان راه‌حلّی کارآمد برای مقابله با چنین مشکلی استفاده می‌شود و راهکارهای متفاوتی برای درج این نقاط ارائه شده است که به دو دسته کلی ایستا و پویا تقسیم می‌شوند.

در روش ایستا نقاط ذخیره وضعیت برنامه، بطور دوره‌ای (زمانی و یا مکانی) درج می‌شود [1]. در روش پویا، این نقاط با استراتژی‌های مختلفی در کد قرار می‌گیرد [2]. در پردازنده‌هایی که حافظه اصلی آنها غیرفرّار است، اگر متغیری از یک خانه از حافظه غیرفرّار، بارگیری[8] شود و سپس مقدار جدیدی در همان خانه از حافظه ذخیره شود[9]؛ و در این بین نقاط ذخیره وضعیت برنامه، وجود نداشته باشد، با کاهش مقدار انرژی ورودی و از دست رفتن وضعیت برنامه، امکان ایجاد مشکل ناسازگاری و عدم پیشرفت رو به جلو وجود دارد. مجموعه‌ی دستورالعمل‌های ذخیره کردن پس از بارگیری، یکی از فاکتورهایی است که باعث نمود مشکل ناسازگاری در برنامه می‌شود.

۱-۱- تاریخچه‌ی کارهای مشابه

در این بخش، ابتدا بطور مختصر کارهای مرتبط در مورد سیستم‌های برداشت انرژی را ارائه می‌دهیم. سپس کارهای موجود در مورد مدل‌های مختلف درج نقاط ذخیره وضعیت برنامه در سیستم‌های برداشت انرژی را توضیح می‌دهیم و آخرین رویکردها را برای حل مشکل ناسازگاری، مورد بحث قرار می‌دهیم و در نهایت توضیحی پیرامون نزدیک‌ترین پژوهش به این پژوهش را ارائه می‌دهیم.

۱-۱-۱- سیستم‌های برداشت انرژی

منابع ناپایدار انرژی از قبیل خورشید، حرکت، رادیوفرکانس، تابش‌های الکترومغناطیس و گرما می‌توانند انرژی کافی را برای اینکه سیستم تعبیه‌شده، کاملا مستقل باشد، فراهم کنند. برای دستگاه‌های فوق کم مصرف، منابع انرژی با چگالی کم، از قبیل گرمای بدن و میکروسولار می‌توانند انرژی مناسبی را برای راه‌اندازی این دستگاه‌ها فراهم کنند. مرجع [3] یک شبکه میکروسولار حسگرها را طراحی کرده است. مرجع [4]یک ریزپردازنده‌ی غیرفرّار را ارائه داده است، به همراه یک سیستم برداشت انرژی خورشیدی، که حتی تحت تابش کم نور خورشید هم کار می‌کند.

۱-۱-۲- انواع مدل‌های درج نقاط ذخیره وضعیت برنامه

ذات نوسانی انرژی‌های برداشت شده، مانع از پذیرش گسترده‌ی نرم‌افزارهای بزرگ در سیستم‌های تعبیه‌شده‌ای است که با این دسته از انرژی‌ها تغذیه می‌شوند. پژوهشگران در چرخه‌های مختلف انرژی و برای اجرای مستمر برنامه‌ها در این نوع سیستم‌ها، از حافظه‌های غیر فرّار استفاده می‌کنند تا وضعیت میانی و فرّار برنامه‌ها را ذخیره کنند. در مرجع [5] یک سیستم نرم‌افزاری برای دستگاه‌های شناسای رادیویی[10] طراحی شده است که حافظه فلش را برای ذخیره آنی وضعیت برنامه استفاده کرده است. مرجع [6] از حافظه‌ی فروّالکتریکی،[11] برای افزایش کارایی نقاط ذخیره وضعیت برنامه استفاده کرده است؛ در حالی که کارایی دسترسی این حافظه، می‌تواند با استفاده از حافظه‌ی ایستا[12] بیشتر شود[7].

۱-۱-۳- آخرین رویکردها برای حل مشکل ناسازگاری

مرجع [1] با درج نقاط ذخیره وضعیت، در طی برنامه و بطور متوالی، سازگاری داده‌های برنامه را حفظ می‌کند و تضمین می‌کند که داده‌های غیرفرّار در هنگام بازنشانی مجدد، با استفاده از مکانیزم نسخه‌بندی داده[13]، سازگار باقی می‌ماند. این مرجع پیش از درج نقاط ذخیره وضعیت برنامه، یک رونوشت از تمام متغیرهای غیرفرّار که پتانسیل ایجاد خطای ناسازگاری را دارند، در مرز وظیفه ایجاد کرده و برنامه را به چندین بخش قابل اجرا تقسیم‌بندی می‌کند و آنها را توسط این نقاط درج شده به هم می‌چسباند.

راهکار دیگری برای حل مشکل ناسازگاری؛ افزایش حد آستانه‌ی انرژی برای فعال کردن نقاط ذخیره وضعیت برنامه، و سپس در حالت خواب قراردادن سیستم است [6]. اگر این حد آستانه به اندازه کافی بزرگ باشد، همیشه فرآیند ذخیره وضعیت با موفقیت انجام شود و سپس سیستم به وضعیت خواب می‌رود، در نتیجه هیچ‌گونه عقب‌گردی وجود ندارد و بنابراین مشکل ناسازگاری وجود نخواهد داشت. در این وضعیت، محاسبات تا زمانی که ولتاژ زیادی ذخیره نشده باشد، شروع نمی‌شود و در نتیجه، زمان کار نرمال سیستم تا حد زیادی کاهش پیدا می‌کند و اکثر اوقات سیستم در حالت خواب به سر می‌برد تا ولتاژ افزایش پیدا کند. این‌گونه معماری‌ها اگر به جای خواب بعد از ذخیره وضعیت برنامه، اجرای برنامه را ادامه دهند، همیشه مشکل ناسازگاری رخ می‌دهد.

مرجع [8] طرح CACI را ارائه کرده است که با جستجو در کد و یافتن مجموعه دستورالعمل‌های ذخیره کردن پس از بارگیری، نقاط ذخیره وضعیت برنامه را در بین آن مجموعه‌ها درج می‌کند تا هم سربار سیستم را کاهش دهد و هم صحت برنامه و پیشرفت کار را تضمین کند.

۱-۲- هدف از انجام این پژوهش

هدف از انجام این پژوهش برطرف کردن مشکل ناسازگاری و پیشرفت رو به جلو در سیستم‌های نهفته خودکار با استفاده از الگوریتم فرامکاشفه‌ای ژنتیک است. در این رابطه با مدل کردن این سیستم‌ها و ادغام آن با الگوریتم ژنتیک به نتایج مطلوبی رسیده‌ایم که در بخش بعدی بطور مفصل توضیح می‌دهیم.

۲- الگوریتم پیشنهادی

در این بخش، ما طرح درج کارای نقاط ذخیره وضعیت برنامه در سیستم‌های نهفته با قابلیت برداشت انرژی از محیط با استفاده از الگوریتم ژنتیک پیشنهاد می‌نمائیم.

۲-۱- طرح مساله

خطای ناسازگاری در پردازنده‌های غیرفرّار، که وضعیت برنامه را ذخیره می‌کنند، زمانی رخ می‌دهد که، قبل از بارگیری و بعد از بارگیری یک متغیر و ذخیره کردن آن در همان محل از حافظه، یک نقطه ذخیره وضعیت برنامه درج شده باشد و در حین اجرای برنامه، انرژی سیستم تمام شود؛ پس از جمع‌آوری مجدد انرژی، ادامه‌ی فرآیند اجرای برنامه به قبل از دستورالعمل بارگیری بازگردد و مقدار بروز شده‌ی جدیدی را از آن خانه از حافظه غیرفرّار بارگیری کند، و ادامه‌ی برنامه را با این مقدار بروز شده انجام دهد که موجب بروز خطای ناسازگاری می‌گردد.

در این صورت برای رفع این خطا، حتما باید نقاط ذخیره وضعیت برنامه را بین دستورالعمل بارگیری و ذخیره کردن قرار داد. درج نقاط ذخیره و ضعیت برنامه در میان دستورالعمل‌هایی که بطور بالقوه ایجاد خطای ناسازگاری می‌کنند باعث می‌گردد که، در زمان اجرای برنامه و پس از اتمام انرژی، ادامه‌ی اجرای برنامه به نقطه ذخیره وضعیت برنامه بازگردد و چون این نقطه

پس از دستورالعمل بارگیری است؛ بنابراین مقدار جدیدی از حافظه غیرفرّار بارگیری نمی‌شود و خطای ناسازگاری رخ نمی‌دهد.

از این رو با استفاده از الگوریتم ژنتیک محل درج نقاط ذخیره وضعیت برنامه را در بین آن دستورالعمل‌ها تعیین می‌کنیم و در پی آن هستیم که ۱) تمام خطاهای مربوط به ناسازگاری را از بین ببریم؛ ۲) ضمن حفظ صحت برنامه، تعداد نقاط ذخیره برنامه را کاهش دهیم. هدف اول صحت برنامه را تضمین می‌کند و هدف دوم سربار حاصل از درج نقاط اضافی ذخیره وضعیت برنامه را کاهش می‌دهد. در حالی که هدف آخر، سربار زمان کامپایل برنامه را کاهش می‌دهد.

اولین چالش در این مسئله، پیداکردن جفت دستورالعمل‌هایی است که بطور بالقوه ایجاد خطا می‌کنند. چالش دوم این است که نقاط ذخیره وضعیت برنامه را کجای برنامه درج کنیم.

مقدار ثابت L_{max} را بیشینه آستانه فاصله‌ی بین دو نقطه ذخیره وضعیت برنامه در نظر گرفته‌ایم. بیشینه فاصله‌ی مد نظر، روند برنامه را ملزم به حرکت رو به جلو می‌کند. از طرف دیگر، از آنجا که هر دستورالعمل برنامه، مقدار انرژی خاص خود را مصرف می‌کند؛ این فاصله، مجموع انرژی مصرفی دستورالعمل‌های اجرا شده تعریف می‌گردد و در انتهای این فاصله، یک نقطه ذخیره وضعیت برنامه درج می‌شود.

بنابراین در این بخش، ما طرح دو مرحله‌ای درج کارای نقاط ذخیره وضعیت برنامه در سیستم‌های نهفته با قابلیت برداشت انرژی از محیط با استفاده از الگوریتم ژنتیک را پیشنهاد می‌نمائیم تا این مشکل را برطرف کنیم. در مرحله اول جفت دستورالعمل‌های بارگیری و ذخیره کردن را که خطاهای بالقوه را در برنامه ایجاد می‌کنند، پیدا می‌کنیم. مرحله دوم با استفاده از الگوریتم ژنتیک پیشنهادی نقاط ذخیره وضعیت برنامه را در قطعه کد قرار می‌دهیم.

۲-۲- راه حل پیشنهادی

الگوریتم ژنتیک از طبیعت الهام گرفته شده است و راه‌حل‌ها به شکل رشته یا سایر ساختارهای داده‌ای رمزگذاری می شوند (ژن‌ها). الگوریتم ژنتیک با یک استخر (جمعیت) اولیه‌ای از ژن‌ها کار خود را آغاز می‌کند، که بطور تصادفی یا با روشی مکاشفه‌ای تولید می‌شوند. این جمعیت بطور تکراری، برای یافتن ژن بهینه و یا ژن نزدیک به بهینه تکامل می‌یابد. برای این هدف سه عمل که از طبیعت تقلید شده است، انجام می‌شود: انتخاب، ترکیب و جهش.

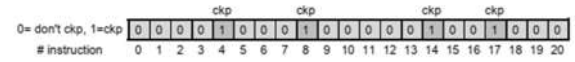
در عمل انتخاب، بعضی از ژن‌ها انتخاب می شوند تا در نسل بعدی زنده باشند و ژن‌های جدیدی را تولید کنند. در عمل ترکیب، دو یا چند ژن از ژن‌های انتخاب شده برگزیده می‌شوند تا ژن‌های جدید تولید شوند، و در نهایت در عمل جهش، یک یا تعداد بیشتری از ژن‌های انتخابی برای حفظ تنوع در جمعیت جهش می‌یابند. الگوریتم ژنتیک زمانی که شرط خاتمه کار برآورده شود به پایان می‌رسد، برای مثال زمانی که حداکثر تعداد تکرار الگوریتم اجرا شود.

Algorithm 1: Fitness Function Algorithm

```
1  Input: Solution sol_i[ ],LS_pairs
2  Output: Fitness of sol_i[ ]
3  function FitnessFunction(sol_i[ ])
4      Data: set Fitness to zero, IsFeasible to True,
5      Penalty_Lmax = 5000, Penalty_Consistency =
       100000, Penalty_Checkpoint = 10, N_exe = 10, L_max = 15
6      foreach LS_pairs do
7          if ld_BB = st_BB then
8              if ld_pos less than st_pos then
9                  Penalty_consistency(Paths, sol_i[ ])
10             else
11                 foreach Paths from ld_pos to st_pos do
12                     Penalty_consistency(Paths, sol_i[ ])
13         else
14             foreach Paths from ld_BB to st_BB do
15                 Penalty_consistency(Paths, sol_i[ ])
16     foreach inst from start to end do
17         Increment Fitness By (sol[inst] * Penalty_Checkpoint) * N_exe
18         max path = DFS(inst) for return maximum path
19         if maxpath > L_max then
20             Increment Fitness By Penalty_Lmax
21             set IsFeasible to False
22     return Fitness of sol_i[ ]
23 function Penalty_Consistency(Paths, sol_i[ ])
24     if paths HAS NOT checkpoint then
25         Increment Fitness By Penalty_Consistency
26         set IsFeasible to False
27     else
28         continue
29     return Fitness of sol_i[ ]
```

الگوریتم(۱): تابع برازش

برای طراحی یک الگوریتم ژنتیک که مشکل ناسازگاری در سیستم‌های نهفته با قابلیت برداشت انرژی از محیط را برطرف نماید، باید راهی را مشخص کنیم تا بتوانیم ژن‌ها را کدگزاری کنیم. از آنجا که برای حل این مشکل نیاز است تا بدانیم در کدام قسمت از کد باید نقاط ذخیره وضعیت برنامه درج شود، همانند شکل (۱) از یک آرایه به طول بیشترین تعداد خطوط برنامه به عنوان ژن استفاده می‌کنیم و آن را فامتن[۱۴] می‌نامیم. در هر کدام از خانه‌های این آرایه یک مقدار دودویی قرار می‌گیرد، که مقدار صفر به منزله عدم درج نقطه ذخیره وضعیت در انتهای دستورالعمل متناظر در کد برنامه و مقدار یک به منزله قرارگیری نقطه ذخیره وضعیت برنامه در انتهای آن دستورالعمل است.



شکل (۱): فامتن تولید شده با الگوریتم ۲ برای برنامه‌ای با طول ۲۱ دستورالعمل

پس از تولید یک ژن توسط الگوریتم ژنتیک پیشنهادی بایستی راهی پیدا کنیم تا برازش[۱۵] آن ژن را محاسبه نمائیم. به این منظور با استفاده از الگوریتم(۱) اقدام به محاسبه برازش ژن‌ها می‌کنیم.

این الگوریتم به عنوان ورودی، یک فامتن (Solution $sol_i[]$) و محل خطاهای بالقوه (LS_{pairs}) را دریافت می‌کند و به عنوان خروجی برازش فامتن ورودی را محاسبه و برمی‌گرداند. در ابتدا، الگوریتم متغیرهایی را مقداردهی می‌کند (خط ۴ و ۵). این متغیرها به ترتیب عبارت هستند از: برازش، که مقدار اولیه صفر می‌گیرد، امکان‌پذیری[۱۶] فامتن که به عنوان مقدار اولیه True می‌گیرد. این مقدار به منزله این است که، این فامتن خطای ناسازگاری ندارد و یا مقدار بیشینه آستانه فاصله بین نقاط ذخیره وضعیت برنامه L_{max} را رعایت کرده است، در غیر اینصورت مقدار False می‌گیرد. سه مقدار ثابت هم به عنوان جریمه (پنالتی) به ترتیب برای خطای L_{max}، خطای ناسازگاری و به ازای هر کدام از نقاط ذخیره وضعیت برنامه که در فامتن باشد؛ یک جریمه

ناچیز Checkpoint در نظر گرفته شـده اسـت. همچنین مقادیر ثابت تعداد اجرای حلقه‌های برنامه N_{exe} و L_{max} نیز تعیین می‌گردند.

در ادامه، کارهای زیر را برای هریک از خطاهای بالقوه انجام می‌شـود (خط ۶)؛ اگر دسـتورالعمل بارگیری و دسـتورالعمل ذخیره‌کردن در یک بلوک پایه[۱۷] یکسان قرار داشتند و شماره دستورالعمل بارگیری از شماره دستورالعمل ذخیره‌کردن کمتر بود یک بار تابع جریمه ناسازگاری اجرا می‌شود، در غیر این صـورت به ازای تمام مسـیرهای بین دسـتورالعمل بارگیری و دسـتورالعمل ذخیره‌کردن، تابع جریمه ناسازگاری اجرا می‌شود (خطوط ۷ تا ۱۲). در صورتی که دستورالعمل بارگیری و دستورالعمل ذخیره‌کردن در یک بلوک پایه مشابه قرار نداشـتند، به ازای تمام مسـیرهای بین بلوک پایه دسـتورالعمل بارگیری و بلوک پایه دسـتورالعمل ذخیره‌کردن، تابع جریمه ناسـازگاری اجرا می‌شـود (خطوط ۱۳ تا ۱۵).

و اما در ادامه، برای وارد کردن جریمه به فامتن ورودی چنین عمل می‌کنیم (خط ۱۶) که برای تمام دسـتورالعمل‌های برنامه، از ابتدا تا انتها، اگر مقدار ژن متناظر دسـتورالعمل جاری برابر یک بود، به تعداد اجرای آن حلقه به اندازه جریمه Checkpoint به مقدار برازش فامتن افزوده می‌شـود (خط ۱۷). متغیر(maxpath) به عنوان بیشـینه فاصـله تا اولین همسـایه ذخیره وضـعیت برنامه با استفاده از تابع جستجوی اول عمق مقداردهی می‌شود (خط ۱۸) و اگر مقدار این متغیر از مقدار L_{max} بیشتر باشد، مقدار برازش فامتن به اندازه جریمه L_{max} افزایش پیدا می‌کند و خاصـیت امکان‌پذیری آن برابر False می‌گردد (خط ۱۹ تا ۲۱). در انتهای این الگوریتم مقدار برازش فامتن بازگردانده می‌شود (خط ۲۲).

تابع جریمه ناسازگاری به این صورت عمل می‌کند که فامتن و مسیرهای متناظر بین محل خطاهای بالقوه را به عنوان ورودی دریافت می‌کند (خط ۲۳)، اگر مسیرهای ورودی در فامتن متناظر فاقد نقطه ذخیره وضعیت برنامه باشد، سـپس مقدار برازش فامتن به اندازه جریمه ناسـازگاری افزایش پیدا می‌کند و خاصـیت امکان‌پذیری آن برابر False می‌گردد (خط ۲۴ تا ۲۶)؛ در غیر این صـورت تابع به ادامه کار بازمی‌گردد (خط ۲۷ و ۲۸). در انتهای این تابع مقدار برازش فامتن ورودی بازگردانده می‌شود (خط ۲۹).

الگوریتم پیشنهادی ما برای درج کارای نقاط ذخیره وضعیت برنامه و حل مشـکل‌های پیش گفته، در الگوریتم(۲) نشـان داده شـده اسـت. این الگوریتم تعداد جمعیت اولیه و حداکثر تعداد حلقه تکرار را به عنوان ورودی دریافت می‌کند (خط ۱) و بهترین فامتن را تولید می‌کند (یک فامتن با کمترین مقدار برازش در میان تمام فامتن‌های تولید شـده) (خط ۲). الگوریتم با ایجاد یک جمعیت تصـادفی از فامتن‌ها شـروع می‌شـود (خط ۴) و مقدار برازش آنها را با استفاده از الگوریتم(۱) محاسبه می‌کند (خط ۵). سپس، روال زیر بطور تکراری تا رسـیدن به حداکثر تعداد حلقه تکرار انجام می‌شـود. ابتدا، فامتن‌ها به ترتیب کمترین مقدار برازش مرتب می‌شـوند (خط ۸)، و یک فامتن با کمترین مقدار برازش انتخاب می‌شود (خط۹).

Algorithm 1: Genetic Algorithm to insert checkpoint in solution

```
1  Input: Max Iteration, population Size
2  Output: A genotype leading to minimum Fitness among all visited
           genotypes
3  function Genetic Algorithm
4      Initialize(population);
5      Compute Fitness(population);
6      iter ← 0 ;
7      while iter < Maxiter do
8          Sort(population);
9          current_best ← GetBest(population);
10         if current_best.fitness < best.fitness then
11             best ← current_best;
12         Selection(population, α);
13         Crossover(population);
14         Mutate(population, β);
15         Compute Fitness(population);
16         iter ← iter + 1;
17     return best;
```

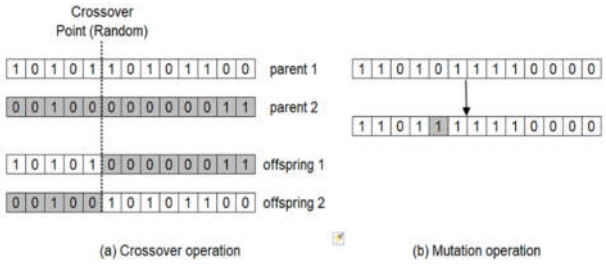
الگوریتم(۲): الگوریتم ژنتیک پیشنهادی

سـپس بررسـی می‌شـود تا مشـخص شـود که آیا برازش آن از بهترین فامتنی که قبلا پیدا شده بهتر (کمتر) است یا خیر؟

اگر چنین اسـت، این فامتن به عنوان فامتن برتر جدید انتخاب می‌شـود (خط۱۰ و ۱۱). پس از آن، مقداری (α) از بهترین ژن‌ها انتخاب می‌شـوند تا در مرحله بعد زنده بمانند و مابقی فامتن‌ها کشته می‌شوند(خط۱۲).

ما از انتخاب نخبه‌گرا اسـتفاده می‌کنیم، به این معنا که بهترین ژن‌ها در نسل حاضر برای حضور در نسل بعد حفظ می شوند؛ بعد از پرو سه انتخاب، عملیات ترکیب بر روی فامتن‌های زنده انجام می‌شـود تا فامتن‌های جدیدی برای مرحله بعد به جای فامتن‌های کشته شده تولید شود (خط ۱۳). سپس، در نسـل جدید فامتن‌ها با احتمال (β) جهش پیدا می‌کنند تا از همگرایی سـریع الگوریتم به سـوی کمینه محلی جلوگیری شـود (خط۱۴). این کارها تا زمان پایان حلقه (بیشـترین تعداد تکرار) تکرار می‌شـود. این الگوریتم با بازگرداندن بهترین فامتن به پایان می‌رسد.

در شـکل (۲) بخش a یک نمونه از عمل ترکیب که در الگوریتم ژنتیک پیشنهادی استفاده شده است، نمایش داده می‌شود. همانطور که در این شکل نشـان داده شـده اسـت ما از دو فامتن والد با یک نقطه برای عمل ترکیب اسـتفاده می‌کنیم؛ به این معنی که برای هر عمل ترکیب دو ژن والد انتخاب می‌شوند، و یک نقطه در بین این فامتن‌ها به صورت تصادفی انتخاب می‌شود و تمام اطلاعات یک طرف آن نقطه در هر دو ژن تعویض می‌شـود. در نتیجه، دو ژن جدید (فرزندان) تولید می‌شـوند و این فرزندان به جمعیت بعدی اضـافه می‌شـوند. در شـکل (۲) بخش b نمونه‌ای از عمل جهش مورد اسـتفاده در الگوریتم پیشنهادی را نشان می‌دهد. در عمل جهش، یک داده از یک ژن بطور تصـادفی انتخاب می‌شـود و مقدار آن به یک مقدار مخالف مقدار حاضـر تغییر می‌یابد.



شکل (۲): عمل ترکیب و جهش

۳-۲- ارزیابی تجربی

برای ارزیابی روش‌های پیشنهادی، ما محک[۱۸] جستجوی خطی[۱۹] را استفاده کردیم. جدول (۲-۱) توصیف‌ها و اندازه محک مورد استفاده را نشان می‌دهد.

جدول (۲-۱): مشخصات محک جستجوی خطی

مشخصه	مقدار
تعداد خطوط کد اسمبلی	۴۳ خط
تعداد بلوک‌های پایه	۷ بلوک
تعداد دستورالعمل‌های ذخیره کردن پس از بارگیری	۸ جفت
میانگین اجرای هر بلوک پایه	۴,۱۴ دور
تعداد دستورالعمل‌های دسترسی به حافظه	۱۷ دستورالعمل
سایر دستورالعمل‌ها	۲۶ دستورالعمل
انرژی مصرفی دستورالعمل‌های دسترسی به حافظه	۲ NJ
انرژی مصرفی سایر دستورالعمل‌ها	۱ NJ
انرژی مصرفی هر نقطه ذخیره وضعیت برنامه	۵ NJ

برای یافتن مقدار مناسب برای هر مشخصه الگوریتم پیشنهادی، با اجرای چندین آزمایش مقادیر مختلفی را برای هر مشخصه بررسی کردیم. نتایج به دست آمده به شرح زیر است:

نرخ انتخاب (α): نتایج آزمایش‌های ما نشان داد که تنظیم $\alpha = 0.5$ منجر به تولید نتایج بهتری نسبت به سایر موارد مورد بررسی می شود. توجه داشته باشید که برای نرخ انتخاب بسیار بالا (به عنوان مثال، ۹.۰)، فرزندان بسیار کمی تولید می‌شوند و از این رو فضای راه‌حل پایین‌تری کاوش می‌شود. از طرف دیگر، برای نرخ انتخاب پایین‌تر (به عنوان مثال، ۰,۱)، بسیاری از راه‌حل‌های خوب را که می‌تواند برای تولید فرزندان بهتر استفاده شود، از دست خواهیم داد.

نرخ جهش (β): نتایج آزمایش‌ها نشان داد که تنظیم $\beta = 0.1$ بهتر از سایر موارد مورد بررسی است. توجه داشته باشید که برای نرخ جهش بسیار زیاد (به عنوان مثال، ۰,۹)، الگوریتم پیشنهادی به یک الگوریتم جستجوی تصادفی تبدیل می‌شود و برای نرخ جهش بسیار پایین (به عنوان مثال، ۰,۰۰۱)، الگوریتم به سرعت در حداکثر مقدار محلی همگرا می‌شود.

اندازه جمعیت: تنظیم اندازه جمعیت روی عدد۱۰۰۰، باعث می‌شود که بین زمان اجرای الگوریتم و کیفیت راه‌حل‌ها نسبت به سایر موارد، تقاطع بهتری انجام شود. توجه داشته باشید که برای اندازه‌های جمعیتی بسیار کم(به عنوان مثال (۱۰)، تنوع ژن‌ها در جمعیت کاهش می‌یابد و از این رو فضای جواب پایین‌تر کاوش می‌شود. از طرف دیگر، برای اندازه‌های بالاتر جمعیت، الگوریتم کند می‌شود.

حداکثر تکرار مجاز (Max iter): با توجه به نتایج آزمایش ها، ما $\text{Max}_{\text{iter}} = 300$ را تعیین می‌کنیم. توجه داشته باشید که برای حداکثر تکرار (به عنوان مثال، ۵۰)، کیفیت راه‌حل‌ها به میزان قابل توجهی کاهش می‌یابد. از طرف دیگر، برای حداکثر تکرار (به عنوان مثال، ۱۰۰۰)، زمان اجرای الگوریتم بطور قابل توجهی بدون هیچ‌گونه پیشرفت چشمگیری در راه‌حل‌ها افزایش می‌یابد.

کد محک جستجوی خطی با استفاده از مترجم[۲۰] ((arm-gcc)) ترجمه شده است؛ در ادامه کد هم‌گذاری[۲۱] تولید شده به گراف روند اجرای برنامه[۲۲] تبدیل شده که این گراف به عنوان ورودی به الگوریتم(۱) داده شده است. پس از اجرای الگوریتم، یک ژن مشابه شکل (۱) تولید می‌شود که در هر خانه از آن که مقدار باینری یک قرار بگیرد به منزله درج نقطه ذخیره وضعیت برنامه در انتهای آن دستورالعمل است.

جدول (۲-۲) مشخصات کامل گراف تولید شده از محک مورد نظر را نشان می‌دهد؛ تعداد کل دستورالعمل‌های برنامه ۴۳ عدد است که از این تعداد، ۱۷ دستورالعمل مربوط به دسترسی به حافظه است و ۲۶ دستورالعمل مربوط به سایر دستورالعمل‌ها است. مقدار N_{exe} نیز با توجه به تعداد اجرای حلقه‌های محک به صورت دستی وارد شده است. واحد اندازه‌گیری انرژی در جدول‌ها نانو ژول است.

انرژی مصرفی هر بلوک پایه، حاصل ضرب تعداد اجرای آن بلوک پایه در مجموع دستورالعمل‌های هر بلوک و دستورالعمل‌های دسترسی به حافظه در آن بلوک است. (با فرض اینکه هر دستورالعمل دسترسی به حافظه دو واحد انرژی مصرف می‌کند، این دستورالعمل‌ها دوبار محاسبه شده است).

انرژی مصرفی نقاط ذخیره درج شده در هر بلوک پایه حاصل ضرب سه مشخصه است: تعداد اجرای آن بلوک، تعداد نقاط ذخیره وضعیت برنامه در آن بلوک و انرژی مصرفی هرکدام از نقاط ذخیره وضعیت برنامه که در اینجا مقدار پنج در نظر گرفته شده است. در نهایت، انرژی مصرفی کل هر بلوک پایه حاصل جمع انرژی مصرفی آن بلوک و انرژی مصرفی نقاط ذخیره وضعیت انرژی آن بلوک است.

جدول (۲-۲): مشخصات بلوک‌های پایه

شماره بلوک پایه	تعداد دستورالعمل‌های هر بلوک پایه (دستورالعمل)	تعداد دستورالعمل‌های دسترسی به حافظه در هر بلوک پایه (دستورالعمل)	تعداد اجرای هر بلوک پایه	انرژی مصرفی هر بلوک پایه (NJ)	تعداد نقاط ذخیره وضعیت برنامه با الگوریتم CACI در هر بلوک پایه	تعداد نقاط ذخیره وضعیت برنامه با الگوریتم ژنتیک در هر بلوک پایه	انرژی مصرفی نقاط ذخیره وضعیت برنامه درج شده با الگوریتم CACI در هر بلوک پایه	انرژی مصرفی نقاط ذخیره وضعیت برنامه درج شده با الگوریتم ژنتیک در هر بلوک پایه	انرژی مصرفی کل هر بلوک پایه با الگوریتم CACI	انرژی مصرفی کل هر بلوک پایه با الگوریتم ژنتیک
۰	۱۹	۱۰	۱	۲۹	۳	۴	۱۵	۲۰	۴۴	۴۹
۱	۹	۳	۹	۱۰۸	۱	۱	۴۵	۴۵	۱۵۳	۱۵۳
۲	۳	۲	۸	۴۰	۱	۱	۴۰	۴۰	۸۰	۸۰
۳	۵	۲	۹	۶۳	۱	۰	۴۵	۰	۱۰۸	۶۳
۴	۱	۰	۰	۰	۱	۱	۰	۰	۰	۰
۵	۲	۰	۱	۲	۱	۰	۵	۰	۷	۲
۶	۴	۰	۱	۴	۰	۰	۰	۰	۴	۴
جمع	**۴۳**	**۱۷**		**۲۴۶**	**۸**	**۷**	**۱۵۰**	**۱۰۵**	**۳۹۶**	**۳۵۱**

۴-۲- مقایسه نتایج

در این بخش الگوریتم ژنتیک پیشنهادی را با الگوریتم CACI مورد بررسی قرار دادیم که نتایج آن در جدول (۲-۳) بیان شده است. مقدار ثابت L_{max} در این آزمایش ۱۵ در نظر گرفته شده است.

جدول (۲-۳): مقایسه الگوریتم ژنتیک پیشنهادی با الگوریتم CACI

مشخصه‌ها	الگوریتم CACI	الگوریتم پیشنهادی
تعداد نقاط ذخیره وضعیت درج شده	۸ نقطه	۷ نقطه
تعداد اجرای نقاط ذخیره وضعیت برنامه	۳۰ بار	۲۱ بار
عدم صلاحیت (تابع برازش) (مقدار کمتر مطلوب است)	۳۰۰	۲۱۰
خطای رفع نشده ناسازگاری	ندارد	ندارد
تعداد اجرای کامل در ۱۰۰۰۰۰ سیکل	۲۱۵ دور	۲۴۶ دور
انرژی تلف شده در ۱۰۰۰۰۰ سیکل	۹۳۲۴ NJ	۱۰۱۵۸ NJ
انرژی تلف شده در هر دور	۴۳,۳۷ NJ	۴۱,۲۹ NJ
انرژی مصرف شده در هر دور بدون انرژی تلف شده	۳۹۶ NJ	۳۵۱ NJ
کل انرژی مصرف شده در ۱۰۰۰۰۰ سیکل	۹۴۴۶۴ NJ	۹۶۵۰۴ NJ
انرژی مصرف شده در هر دور با انرژی تلف شده	۴۳۹ NJ	۳۹۲ NJ
مصرف انرژی برای ۲۱۵ دور	۹۴۴۶۴ NJ	۸۴۳۴۲ NJ
درصد کاهش انرژی مصرفی در هر دور اجرا با الگوریتم ژنتیک		**۱۰,۷**
درصد کاهش انرژی تلف شده در هر دور اجرا با الگوریتم ژنتیک		**۴,۸**

از نتایج به دست آمده، قابل مشاهده است که تعداد نقاط ذخیره وضعیت درج شده با الگوریتم پیشنهادی ۱۲,۵ درصد کاهش داشته است. در حالی که تعداد اجرای آن نقاط و میزان عدم صلاحیت آنها با ۳۰ درصد کاهش روبرو شده است و در هر دو الگوریتم تمامی خطاهای نا سازگاری موجود رفع شده است؛ از سویی دیگر، پیشرفت رو به جلوی برنامه ۱۴,۴۱ درصد، در یک سیکل افزایش داشته و با این وجود، انرژی مصرفی و انرژی تلف شده در هر دور اجرای کامل به ترتیب ۱۰,۷ و ۴,۸ درصد کاهش را تجربه کرده است. تشریح سایر پارامترهای جدول (۲-۳) در این مقاله نمی‌گنجد.

۳- نتیجه گیری

در این مقاله، ما یک الگوریتم ژنتیک پیشنهاد داده‌ایم که مشکل ناسازگاری داده‌ها را در پردازنده‌های غیرفرّار سیستم‌های برداشت انرژی نهفته با درج کارای نقاط ذخیره وضعیت، حل می‌کند. اولین نوآوری علمی ما در این مقاله مدل کردن سیستم‌های نهفته با قابلیت برداشت انرژی از محیط است و نوآوری علمی بعدی ما نیز، ارائه یک روش جدید با استفاده از الگوریتم ژنتیک است که نقاط ذخیره وضعیت برنامه را در سیستم‌های نهفته با برداشت انرژی بطور کارا درج می‌کند. آخرین نوآوری علمی ما در این مدل، محاسبه میزان ناسازگاری و عدم رعایت هدف پیشرفت رو به جلوی سایر روش‌ها با استفاده از الگوریتم جدیدی است که ارائه داده‌ایم.

نتایج آزمایش‌های ما نشان می‌دهد که الگوریتم ژنتیک پیشنهادی تعداد نقاط ذخیره وضعیت درج شده، تعداد اجرای آن نقاط، انرژی مصرفی و انرژی تلف شده را کاهش می‌دهد و تمام خطاهای ناسازگاری را رفع می‌کند و همزمان پیشرفت رو به جلوی برنامه را افزایش می‌دهد.

برای بهبود بیشتر پیشرفت رو به جلو، در کارهای آینده می‌کوشیم تا الگوریتم را به نحوی گسترش دهیم که، استخر جمعیت اولیه ، فاقد فامتن‌های نامطلوب باشد؛ مقادیر مناسب برای ثابت‌های در نظر گرفته شده در الگوریتم، مانند L_{max} و N_{exe} را جستجو نموده و مقدار بهینه را بدست آوریم و نتایج ارزیابی‌ها را با سایر الگوریتم‌ها، در شرایط متفاوت ارائه کنیم، تا پژوهشگران بهتر بتوانند عملکرد کار ما را در شرایط کلی‌تر بررسی کنند.

همچنین پژوهشگران می‌توانند با استفاده از روش‌های دیگر مانند برنامه‌ریزی خطی، مسئله پیداکردن محل درج نقاط ذخیره وضعیت برنامه را بهینه نمایند. این امر باعث ایجاد فرصت‌هایی برای ایجاد یک سیاست تخصیص حافظه می‌شود که تجارت را در بر می‌گیرد.

سپاسگزاری

در تهیه این مقاله از تجربه‌های دکتر عبدالرضا رسولی کناری و راهنمایی‌ها، ویرایش‌های فنی و بازخوانی‌های آقای دکتر مرتضی محجّل کفشدوز استفاده شده است، که از زحمت‌های آنان سپاسگزاری می‌شود.

مراجع

[1] Lucia, B., and Ransford, B.: ‘A simpler, safer programming and execution model for intermittent systems’, ACM SIGPLAN Notices, 2015, 50, (6), pp. 575-585

[2] Balsamo, D., Weddell, A.S., Merrett, G.V., Al-Hashimi, B.M., Brunelli, D., and Benini, L.: ‘Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems’, IEEE Embedded Systems Letters, 2014, 7, (1), pp. 15-18

[3] Taneja, J., Jeong, J., and Culler, D.: ‘Design, modeling, and capacity planning for micro-solar power sensor networks’, in Editor (Ed.)^(Eds.): ‘Book Design, modeling, and capacity planning for micro-solar power sensor networks’ (IEEE, 2008, edn.), pp. 407-418

[4] Wang, C., Chang, N., Kim, Y., Park, S., Liu, Y., Lee, H.G., Luo, R., and Yang, H.: ‘Storage-less and converter-less maximum power point tracking of photovoltaic cells for a nonvolatile microprocessor’, in Editor (Ed.)^(Eds.): ‘Book Storage-less and converter-less maximum power point tracking of photovoltaic cells for a nonvolatile microprocessor’ (IEEE, 2014, edn.), pp. 379-384

[5] Ransford, B., Sorber, J., and Fu, K.: ‘Mementos: System support for long-running computation on RFID-scale devices’, in Editor (Ed.)^(Eds.): ‘Book Mementos: System support for long-running computation on RFID-scale devices’ (2011, edn.), pp. 159-170

[6] Jayakumar, H., Raha, A., and Raghunathan, V.: ‘QuickRecall: A low overhead HW/SW approach for enabling computations across power cycles in transiently powered computers’, in Editor (Ed.)^(Eds.): ‘Book QuickRecall: A low overhead HW/SW approach for enabling computations across power cycles in transiently powered computers’ (IEEE, 2014, edn.), pp. 330-335

[7] Shiga, H., Takashima, D., Shiratake, S.-i., Hoya, K., Miyakawa, T., Ogiwara, R., Fukuda, R., Takizawa, R., Hatsuda, K., and Matsuoka, F.: ‘A 1.6 GB/s DDR2 128 Mb chain FeRAM with scalable octal bitline and sensing schemes’, IEEE Journal of Solid-State Circuits, 2009, 45, (1), pp. 142-152

[8] Xie, M., Zhao, M., Pan, C., Hu, J., Liu, Y., and Xue, C.J.: ‘Fixing the broken time machine: Consistency-aware checkpointing for energy harvesting powered non-volatile processor’, in Editor (Ed.)^(Eds.): ‘Book Fixing the broken time machine: Consistency-aware checkpointing

for energy harvesting powered non-volatile processor' (2015, edn.), pp. 1-6

پانویس ها

[1] Energy harvesting
[2] Embedded system
[3] Non-Volatile Processor (NVP)
[4] Checkpoint
[5] Consistency
[6] Forward progress
[7] Offline analysis
[8] Load
[9] Store
[10] RFID
[11] FRAM
[12] SRAM
[13] Data versioning
[14] Chromosome
[15] Fitness
[16] IsFeasible
[17] Basic block
[18] Benchmark
[19] Linear search
[20] Compiler
[21] Assembly code
[22] Control flow graph